Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3466

Verwaltung von Testinformationen in der Moduldokumentation

Davide Casciato

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat. Stefan Wagner
Betreuer/in:	Dipl.-Inf. Ivan Bogicevic
Beginn am:	2013-03-01
Beendet am:	2013-08-31
CR-Nummer:	D.2.1, D.2.2, D.2.5, D.2.7, D.2.9

Kurzfassung

Softwaresysteme werden typischerweise in Module unterteilt. Alle relevanten Daten eines Moduls werden in dessen Moduldokumentation, welche ein Teil der Softwarearchitekturdokumentation darstellt, festgehalten. Zu den Daten zählen unter anderem die Testinformationen des Moduls.

Softwarearchitekturdokumentationen werden ihrerseits üblicherweise über Textverarbeitungswerkzeuge oder Wikis erstellt. Beide Werkzeugklassen sind für diese Aufgabe jedoch nicht uneingeschränkt geeignet. Daher wird in dieser Diplomarbeit ein Werkzeug entwickelt, das auf die Softwarearchitekturdokumentation – insbesondere der Moduldokumentation – zugeschnitten ist. Im Werkzeug können Dokumente und Dokumentteile über Vorlagen erstellt werden. Für die Dokumentteile, in denen die Testinformationen eines Moduls verwaltet werden sollen, wird in dieser Arbeit eine spezielle Vorlage vorgeschlagen und für das Werkzeug umgesetzt. Die Vorlage enthält Abschnitte, in denen alle relevanten Testinformationen entweder direkt angegeben oder referenziert werden können. Des Weiteren wird ein Konzept entworfen und durch das Werkzeug umgesetzt, mit welchem die Beziehungen, die in der Softwarearchitekturdokumentationen existieren, explizit definiert und visualisiert werden können. Ein Beispiel für solche Beziehungen sind die zwischen den referenzierten Testinformationen: Z. B. Testfallprotokoll *P* protokolliert einen Testlauf von Testfall *F*. Das Werkzeug wird abschließend evaluiert.

Abstract

Software systems are typically divided into modules. All relevant data of a module are documented in its module documentation. Among other things, these data include the test information of the module. The module documentation itself is part of the overall software architecture documentation.

Software architectures are usually documented with word processors or wikis. However, both kinds of those tools are not fully sufficient for this task. Therefore, this diploma thesis develops a tool, which is specialized for documenting software architectures and modules. The tool provides functionality to create documents and their parts based on templates. This thesis provides a special template for those parts of the document, which concern the test of modules. The template contains sections for all relevant test information. The test information may be named directly or may be reffered to. Furthermore, the thesis provides a concept to explicitly define and visualize the relations, which exist in the software architecture documentation. An example for such relations are those between the test information which were reffered to, e.g., test protocol *P* protocols a test run of test case *C*. The concept is realized within the tool. Finally, the tool is evaluated.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

API	Application Programming Interface 13
BSI	British Standards Institution 21
CAST	Computer Aided Software Testing 31
CTP	Critical Testing Processes 25
ER	Entity-Relationship 20
ERM	Entity-Relationship Modell 33
GUI	Graphical User Interface 33
HTML	Hypertext Markup Language 38
IEEE	Institute of Electrical and Electronics Engineers 18
ID	Identifikationsnummer 26
ISTE	Institut für Softwaretechnologie 11
ISTQB	International Software Testing Qualifications Board 21
JAXB	Java Architecture for XML Binding 51
JUNG	Java Universal Network/Graph Framework 59
J-PaD	Java Package Documentation 11
MVC	Model-View-Controller 24
PDF	Portable Document Format 38
QSW	Qualitätssicherung und Wartung 70
SDK	Software Development Kit 51
STEP	Systematic Test and Evaluation Process 25
SVN	Apache Subversion 19
UML	Unified Modelling Language 20
UniMoDoc	Universal Module Documenter 13
URL	Uniform Resource Locator 40
UUID	Universally Unique Identifier 53
XML	Extensible Markup Language 36

1 Einleitung

Heutzutage können Softwaresysteme sehr umfangreich werden und eine Vielzahl von verschiedenen Funktionalitäten anbieten. Diese Umstände führen dazu, dass die Systeme bei ihrer Entstehung in handhabbare Module mit definierten Aufgaben gegliedert werden. Die Module werden nach ihrem Entwurf realisiert, getestet und meist in mehreren Iterationen zum Gesamtsystem zusammengesetzt. Die Gliederung und die Module selbst müssen, genauso wie die Architekturentscheidungen, detailliert dokumentiert und die entsprechenden Dokumente kontinuierlich aktualisiert werden. Sie bilden die Grundlage für die Realisierung, den Test, die Weiterentwicklung und die Wartung der Software.

In der Praxis gibt es viele Werkzeuge, die den Architekten in der grafischen Modellierung des Systems unterstützen. Jedoch gibt es nur wenige, welche Unterstützung in der textuellen Dokumentation insbesondere der der Moduldokumentation leisten. Hierfür werden meist Dokumentvorlagen für bekannte Textverarbeitungssysteme verwendet. Diese haben zwei signifikante Probleme. Das erste Problem ist, dass sie den Architekten nur in der Strukturierung der Dokumentation unterstützen. Sie bieten jedoch keine Hilfe bei der Eingabe der Daten (z. B. über Hinweise oder Validierungsfunktionen). Das zweite Problem bezieht sich auf die Speicherung der Dokumente. Werden die Dokumente in einem binären und nicht textbasierten Format gespeichert, erschwert dies die Unterstellung unter Versionsverwaltungssysteme. Zum einen können in Konflikt stehende Versionen eines solchen Dokuments durch das System nicht automatisch zusammengeführt werden. Zum anderen kann der Benutzer die semantischen Unterschiede beim Vergleich der Versionen nur mithilfe von speziellen Werkzeugen erkennen.

Ein möglicher Lösungsansatz für die genannten Probleme ist das am Institut für Softwaretechnologie (ISTE) der Universität Stuttgart entwickelte Werkzeug namens Java Package Documentation (J-PaD). J-PaD dient der integrierten Dokumentation von Modulen der Programmiersprache Java. Bei der Entwicklung von J-PaD wurde angenommen, dass Java-Pakete Module in Java darstellen. Konzipiert und realisiert wurde es als Erweiterung für die Entwicklungsumgebung Eclipse. In J-PaD kann der Benutzer Informationen zu den Paketen eines Java-Projekts über eine grafische Oberfläche verwalten. Gespeichert werden die eingegebenen Informationen in Form von Kommentaren in einer Java-spezifischen Datei (`package-info.java`), welche jeweils einmal pro Paket existiert. `package-info.java`-Dateien können seit Java 1.5 verwendet werden, um beispielsweise ganze Pakete mithilfe von Javadoc zu kommentieren.

Abbildung 1.1 zeigt den Zustand von J-PaD nach dem Öffnen einer solchen Datei. Im Editor-Bereich von Eclipse (rechts in der Abbildung) kann der Benutzer alle Informationen der geöffneten `package-info.java`-Datei über eine grafische Oberfläche verwalten. Diese besteht

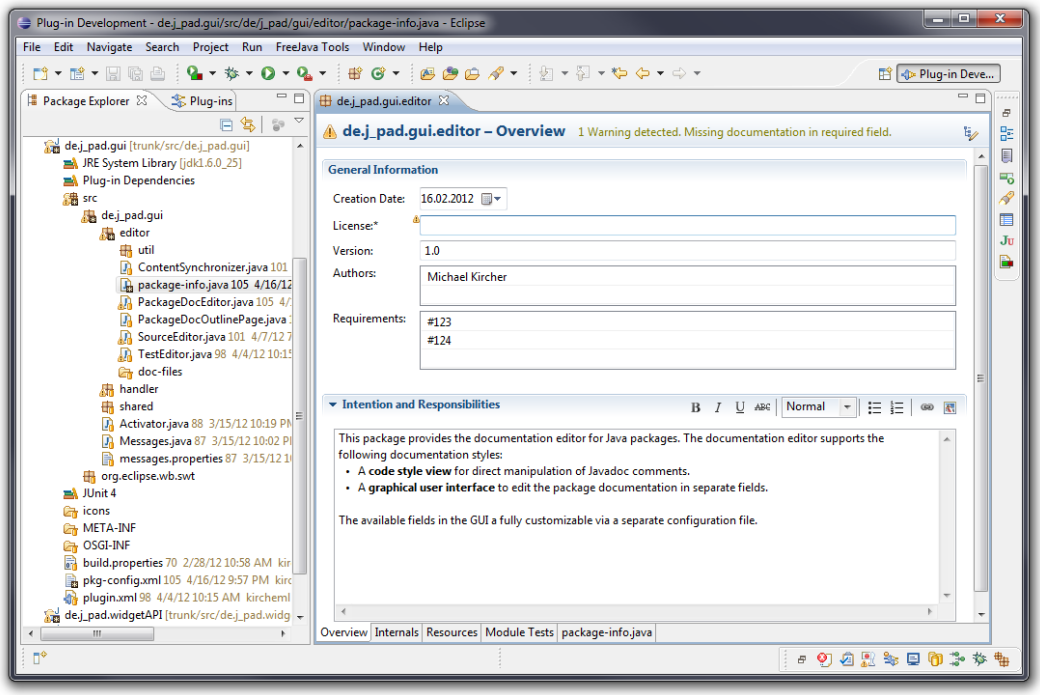


Abbildung 1.1: Screenshot der Modulübersicht von J-PaD

aus unterschiedlichen grafischen Elementen, wie Gruppen und beschrifteten Eingabeelementen. Die Informationen sind in unterschiedliche Kategorien unterteilt, welche durch Reiter am unteren Rand des Editors repräsentiert werden. In der Abbildung ausgewählt ist der Reiter der Modulübersicht, in dem der Benutzer die allgemeinen Informationen des Moduls verwalten kann. Neben diesem existieren weitere Reiter für die Modulinternas, die vom Modul referenzierten Ressourcen und die mit dem Modul verbundenen Testfälle. Über den *package-info.java*-Reiter kann sich der Benutzer die Moduldokumentation auf Programmcodeebene anzeigen lassen und diese manuell bearbeiten.

Die grafische Oberfläche und somit die Struktur der Moduldokumentation kann in einer für das gesamte Java-Projekt geltenden Konfigurationsdatei (`pkg-config.xml`) angepasst werden. Die von J-PaD mitgelieferte Konfiguration definiert eine Oberfläche, welche überwiegend Eingabeelemente für die in [Starke und Hruschka, 2009] vorgeschlagenen Modulinformationen enthält. Abbildung 1.2 zeigt J-PaD nach dem Öffnen einer Konfigurationsdatei. In der Gruppe *Documentation Layout* können die Reiter und deren Inhalte verwaltet werden. Die Gruppe *Widget Settings* ermöglicht die Verwaltung der Details des selektierten Elements der *Documentation Layout*-Gruppe. Für das selektierte Eingabeelement, auch Widget genannt, kann neben dem angezeigten Namen und dem konkreten Type des Eingabeelements eine Abbildung auf ein sogenanntes Tag vorgenommen werden. Die Abbildung ist notwendig, um die in einem Eingabeelement eingegebenen Daten einem Tag eindeutig zuordnen zu können. Die Menge an mitgelieferten Eingabeelementtypen ist beliebig erweiterbar. Hierzu werden

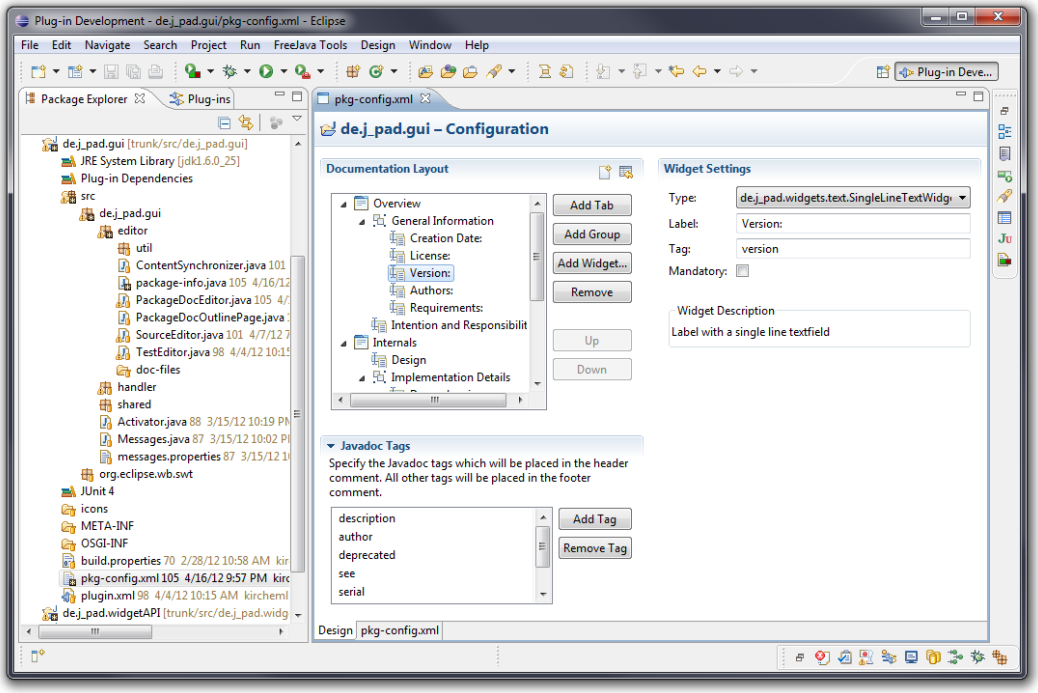


Abbildung 1.2: Screenshot der Konfigurationsverwaltung von J-PaD

neue Java-Klassen, die zur programmatischen Erstellung des Widgets benötigt werden, angelegt und in einem zentralen Widget-Application Programming Interface (API) registriert. Anschließend kann das hinzugefügte Widget in der *Documentation Layout*-Gruppe der grafischen Oberfläche (siehe Abbildung 1.2) verwendet werden. Weiterführende Informationen zu J-PaD und dessen Umsetzung finden sich in [Kircher, 2012] und [Kircher, 2013].

Wie aus dem Namen und der obigen Beschreibung von J-PaD hervorgeht, ist das Werkzeug auf die Dokumentation von Java-Paketen zugeschnitten und verwendet dazu Java-spezifische Konzepte. Um sich von diesen Abhängigkeiten zu lösen, wurde die Entwicklung eines Nachfolgers eingeleitet. Der Nachfolger trägt den Namen Universal Module Documenter (UniMoDoc). Die Konzepte von UniMoDoc und UniMoDoc selbst sollen in dieser und der Arbeit von [Pankratz, 2013] entwickelt werden.

1.1 Motivation

In der Moduldokumentation werden die wichtigsten Informationen zu jedem Modul festgehalten. Zu diesen Informationen gehören die Eigenschaften und die Beziehungen des Moduls zu anderen Modulen. Eine Eigenschaft stellen die für das Modul relevanten Testinformationen, wie z. B. die Testfälle und die Testprotokolle dar. Abhängig vom Benutzer können sie zu verschiedenen Zwecken herangezogen werden. Beispielsweise kann der Architekt, zu dessen Aufgaben nicht nur der Entwurf der Software, sondern auch die Überwachung der Realisierung der Software zählt, durch die Testinformationen feststellen, ob und wie ein Modul getestet wurde.

Die Testinformationen werden in der Moduldokumentation durch Tester verwaltet. Diese erfassen die Testinformationen entweder textuell oder über Referenzen auf entsprechende Testmittel. Daher werden in UniMoDoc Konzepte und Funktionen benötigt, um die Testinformationen zum einen explizit erfassen und zum anderen referenzieren zu können.

J-PaD unterstützt die Verwaltung von Testinformationen vergleichsweise rudimentär. Die Testinformationen können entweder im *Resources*-Reiter (siehe Abbildung 1.4) oder im *Module Tests*-Reiter (siehe Abbildung 1.3) verwaltet werden. In beiden Fällen kann jede Testinformation, dargestellt als Testfall („Test Method“) bzw. als externe Ressource („External Resource“), referenziert werden. Eine über die Referenzierung hinausgehende Verwaltung der Testinformationen ist nicht möglich. Es existieren beispielsweise keine Eingabefelder, um das gewählte Vorgehen beim Test des Moduls textuell zu beschreiben oder die Ansprechpartner des Test in Bezug auf das Modul zu nennen.

1.2 Problemstellung

Die Aufgabe dieser Diplomarbeit ist es daher, die Konzepte und Teile von UniMoDoc zu entwickeln, mit denen die Testinformationen der Moduldokumentation über eine grafische Oberfläche geeignet verwaltet werden können. Die Dokumentation umfasst sowohl die Referenzierung nach extern, als auch die direkte Verwaltung der Testinformationen (z. B. Testvorgehen, Ansprechpartner, ...). Das entwickelte Konzept soll außerdem in UniMoDoc realisiert und evaluiert werden.

Im Rahmen der Aufgabe soll geklärt werden, welche Testinformationen bzgl. eines Moduls potenziell dokumentiert werden können und wie dies geschehen kann. Des Weiteren ist der Stand der Technik in Bezug auf die Thematik der Arbeit zu untersuchen.

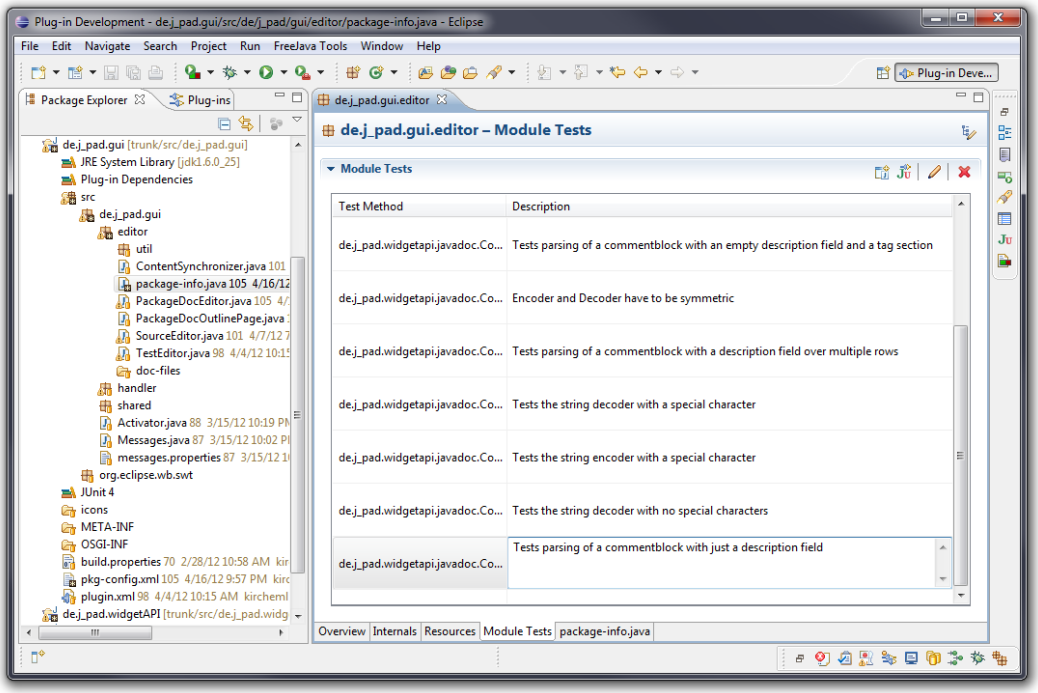


Abbildung 1.3: Screenshot der Testfallverwaltung von J-PaD

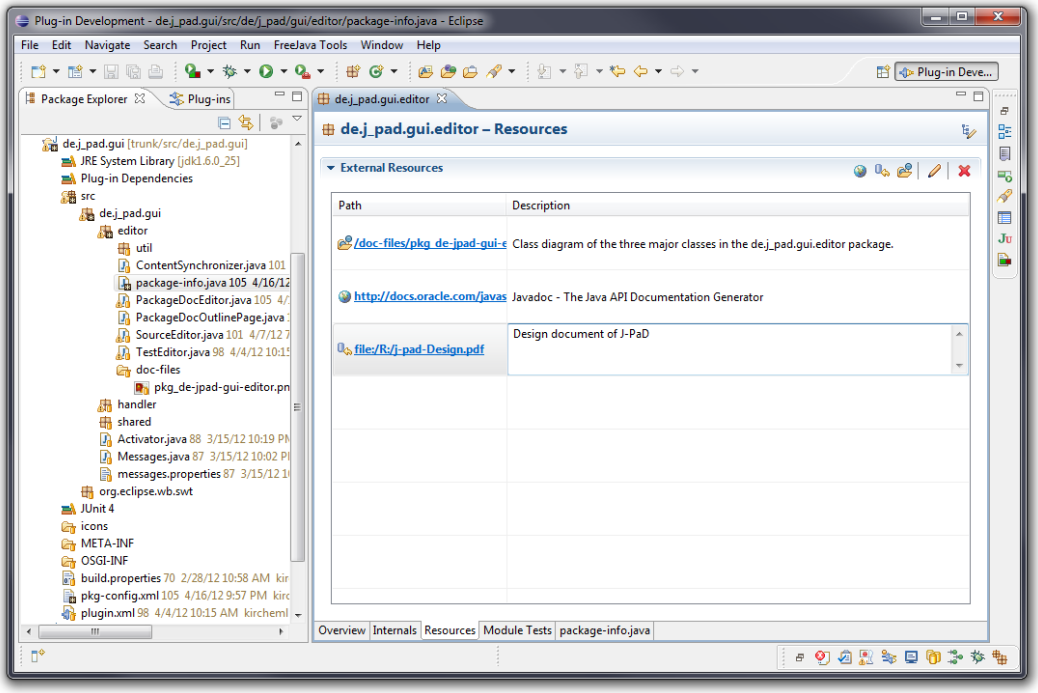


Abbildung 1.4: Screenshot der Verwaltung der externen Ressourcen von J-PaD

1.3 Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 1 – Einleitung leitet in das Thema der Diplomarbeit ein. Hierbei wird die Problemstellung der Arbeit zunächst motiviert und anschließend beschrieben.

Kapitel 2 – Grundlagen und Stand der Technik vermittelt die für die nachfolgenden Kapitel notwendigen Grundlagen.

Kapitel 3 – Anforderungen, Zielgruppe und Konzept beschreibt die Anforderungen an die zu entwickelnde Software und deren Zielgruppe. Anschließend wird das Konzept erläutert, welches der Software zugrunde liegt.

Kapitel 5 – Ergebnisse präsentiert die entstandene Software.

Kapitel 4 – Umsetzung geht auf die technische Umsetzung der Software ein.

Kapitel 6 – Evaluation evaluiert die Software und deren Konzepte.

Kapitel 7 – Zusammenfassung und Ausblick fasst die Diplomarbeit zusammen und schlägt mögliche Verbesserung für die Software vor.

2 Grundlagen und Stand der Technik

In diesem Kapitel werden die theoretischen Grundlagen und der derzeitige Stand der Technik hinsichtlich der Thematik der Diplomarbeit vermittelt. Zunächst wird geklärt, was unter den Begriffen Modul (siehe Kapitel 2.1), Softwarearchitekturdokumentation (siehe Kapitel 2.2), Fehler (siehe Kapitel 2.3) und Test (siehe Kapitel 2.4) in dieser Arbeit verstanden wird. Neben dem Testbegriff wird der der Testinformationen erläutert und der Test von der statischen Prüfung abgegrenzt. Eine Software – und damit auch deren Module – wird auf mehreren sogenannten Teststufen getestet. Die gängigsten werden in Kapitel 2.5 vorgestellt. Die Tests der unterschiedlichen Teststufen finden nicht ungeplant, sondern innerhalb eines parallel zum Vorgehensmodell durchgeführten Testprozesses statt. Ein allgemeiner Testprozess wird in Kapitel 2.6 vorgestellt. Innerhalb der Phasen des Testprozesses entstehen diverse Testdokumente. Es existieren Standards, wie der IEEE-829, die vorschlagen, welche Dokumenttypen mit welchen Inhalten im Testprozess erstellt werden sollen. In Kapitel 2.7 werden die Dokumente des IEEE-829 vorgestellt und mit den zuvor erläuterten Phasen des Testprozesses und den Teststufen in Verbindung gebracht. Abschließend werden die Werkzeuge untersucht, die zur Unterstützung des Testprozesses beitragen können.

2.1 Modulbegriff

Die Bausteine einer Software werden oft als Module, Komponenten oder Units bezeichnet. Alle drei Begriffe werden im Systems and Software Engineering - Vocabulary (IEEE-24765) definiert [IEEE-24765, 2010]. Sie sind jedoch nicht klar gegeneinander abgegrenzt und werden daher in der Praxis oft als Synonyme verwendet. Gemein haben die Definitionen, dass sie einen logisch zusammengehörigen Teil eines Systems identifizieren.

In dieser Arbeit wird entsprechend der obigen Feststellung unter dem Modulbegriff ein logisch zusammengehöriger Teil eines Systems – bestehend aus Software und/oder Hardware – verstanden. Die Zerlegung des Gesamtsystems in Module wird von Softwarearchitekten vorgenommen. Hierbei liegt die logische Zusammengehörigkeit im Ermessen der Architekten. Sinnvolle Möglichkeiten Module zu modellieren sind z. B. die nach dem Zweck oder die nach der technischen Realisierung des Moduls. Ein Modul kann aus beliebigen Bestandteilen eines Systems und somit wiederum aus Modulen bestehen.

Die hier verwendete Definition unterscheidet sich in einem wesentlichen Punkt von der J-PaD zugrundeliegenden. In J-PaD besteht eine direkte Abhängigkeit zur Programmiersprache, da ein Modul mit einem Java-Paket gleichgesetzt wird. In dieser Arbeit hingegen wird ein Modul als frei definierbar betrachtet. Dadurch sind Module prinzipiell unabhängig von

dem verwendeten Programmiermodell und der -sprache. Dies bedeutet am Beispiel der Programmiersprache Java, dass ein Modul ein Paket sein kann, aber nicht zwingend sein muss. In Java können neben Paketen folgende Entitäten als Module angesehen werden: Klassen, OSGi-Bundles [OSGi, 2013], Jigsaw [OpenJDK, 2013] Module. Eine ausführliche Beschreibung der Beispiele und eine detaillierte Auseinandersetzung mit dem Modulbegriff findet sich in [Pankratz, 2013].

2.2 Softwarearchitekturdokumentation

Die Architektur einer Software beschreibt im Allgemeinen seine Struktur, seine Bestandteile, die Schnittstellen nach innen und nach außen und deren Zusammenspiel. Diese Aussage spiegelt sich auch in der Definition des Institute of Electrical and Electronics Engineers (IEEE) wider, welche dieser Arbeit zugrunde liegt:

> **Software architecture** The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.
>
> [IEEE-1471, 2000]

Die Informationen der Softwarearchitektur werden in der Softwarearchitekturdokumentation festgehalten. Sie dient als Plan des Systems, welcher für viele Projektbeteiligte von Interesse ist. Auf ihrer Grundlage können Kunden sicherstellen, dass ihre Anforderungen in einer angemessenen Qualität erfüllt werden, die Entwickler verwenden sie um das System zu entwickeln, und das Testpersonal bezieht aus ihr die Informationen um Testfälle zu erstellen. Der Aufbau und der Inhalt der Softwarearchitekturdokumentation sind nicht standardisiert. Eine weit verbreitete und anerkannte Art der Dokumentation, stellt die über Sichten (engl. views) dar. Die Sichten zeigen die Architektur des Systems aus unterschiedlichen Perspektiven [IEEE-1471, 2000]. Dadurch ist es möglich, bestimmte Aspekte der Architektur innerhalb einer Sicht in den Mittelpunkt zu stellen. In der Literatur finden sich diverse Modelle für die Zusammensetzung und Organisation von Sichten. Diese Arbeit verwendet das Sichtenmodell von [Garlan et al., 2010] als Grundlage. In ihm werden drei Sichten vorgeschlagen: die *Module Views*, die *Component-and-Connector Views* und die *Allocation Views*. Im Folgenden wird nur auf die Module Views eingegangen, da nur sie relevant für diese Arbeit sind.

Module Views Eine Module View kann als Moduldokumentation eines Moduls betrachtet werden. In dieser werden die Beziehungen des Moduls zu anderen Modulen und dessen Eigenschaften festgehalten. Als Module Views wird entsprechend eine Menge dieser Views bezeichnet. Die Beziehungen eines Moduls können unterschiedliche Semantiken besitzen, welche durch entsprechende Beziehungstypen repräsentiert werden. [Garlan et al., 2010] schlagen folgende Beziehungstypen vor: *Is part of*, *Depends on* und *Is a*. Neben den Beziehungen besitzen Module des Weiteren Eigenschaften (engl. properties), wie z. B. den Namen oder den Zweck des Moduls. Eine weitere Eigenschaft stellen Informationen bezüglich des Tests dar. [Garlan et al., 2010] zählen zu diesen wörtlich:

> **Test information** The module's test plan, test cases, test scaffolding, and test data are important to store.
>
> [Garlan et al., 2010]

Eine große inhaltliche Überschneidung zu den Modulen der Module Views, zeigen die Bausteine der Bausteinsicht von [Starke und Hruschka, 2009]. Daher können die in ihr definierten Eigenschaften für Bausteine auf die Module der Module Views übertragen werden. Ebenso wie bei den Modulen der Module Views werden zu den Bausteinen Testinformationen festgehalten. Zu diesen gehören eine Auflistung der wesentlichen Tests des Bausteins und/oder eine Beschreibung des Soll- oder Ist-Zustandes des Tests.

Die Gemeinsamkeit beider Sichtenmodelle hinsichtlich des Tests ist, dass sie vorsehen die wesentlichen Testinformationen des jeweiligen Moduls in dessen Moduldokumentation festzuhalten. Welche Informationen konkret in der Moduldokumentation aufgenommen werden sollen, ist von den Projektbeteiligen zu klären. Nach [Starke und Hruschka, 2009] besitzt eine angemessene Softwarearchitekturdokumentation und damit auch die Moduldokumentation bestimmte Eigenschaften. Beispielsweise soll eine Softwarearchitekturdokumentation *relevant* sein. Das heißt, dass nur solche Informationen in ihr aufgenommen werden, welche für die Architektur relevant sind. Außerdem soll sie *sparsam* sein und keine unnötigen Informationen enthalten. Die von [Starke und Hruschka, 2009] definierten Eigenschaften sind bei der Aufnahme jeglicher Informationen, also auch der der Testinformationen, zu berücksichtigen. Die Verständigung auf die in der Moduldokumentation aufzunehmenden Testinformationen und die Art der Aufnahme wird in der weiteren Ausarbeitung als *Auswahlverfahren der Testinformationen* bezeichnet. Das Auswahlverfahren kann prinzipiell für jedes einzelne Modul, für eine Menge von gleichartigen Modulen, oder für alle Module des Systems unabhängig ihrer Eigenheiten durchgeführt werden.

Techniken und Werkzeuge

[Starke und Hruschka, 2009] haben in vielen ihrer Beratungsprojekten beobachtet, dass in diesen viel Zeit und Aufwand investiert wurde, um die Struktur der Softwarearchitekturdokumentation zu definieren. Daher empfehlen sie und andere Autoren (z. B. [Garlan et al., 2010]) die Verwendung von klar vordefinierten Gliederungsstrukturen, auch Vorlagen oder Templates genannt, als Grundlage für die Softwarearchitekturdokumentation. Durch die klare Struktur wird laut [Starke und Hruschka, 2009] die Verständlichkeit und Navigierbarkeit sowohl bei den Autoren, als auch bei den Lesern des Dokuments gesteigert.

Typischerweise werden solche Vorlagen in gängigen Textverarbeitungsformaten angeboten. Die Erstellung der Dokumentation mithilfe von Textverarbeitungswerkzeugen hat jedoch diverse Nachteile. Zum einen werden die Ausgabedateien der Textverarbeitungswerkzeuge in binären Formaten abgespeichert. Dieser Umstand erschwert die Unterstellung der Softwarearchitekturdokumentation unter eine computergestützte Versionsverwaltung, wie beispielsweise Apache Subversion (SVN). Zum anderen ist die vorgegebene Gliederungsstruktur auf Überschriften und Platzhalter für Freitext beschränkt, was den Benutzer nur

bedingt bei der Eingabe von Daten unterstützt. Des Weiteren können die für die Softwarearchitekturdokumentation relevanten Beziehungen (z. B. zwischen den Modulen) nur durch einfache Hyperlinks ohne jegliche Semantik ausgedrückt werden.

Eine weiteren Werkzeugtyp, mit welchem Softwarearchitekturen dokumentiert werden können, sehen [Starke und Hruschka, 2009] und [Garlan et al., 2010] in Wikis. Diese haben den Vorteil, dass mehrere Benutzer gleichzeitig an den Seiten des Wikis arbeiten können und die Versionsverwaltung in vielen Fällen mitgeliefert wird. Semantische Wikis bieten darüber hinaus die Möglichkeit, Hyperlinks mit Annotationen zu versehen. Mithilfe dieser Funktion können die Beziehungen in der Softwarearchitekturdokumentation geeignet definiert werden. Genauso wie Textverarbeitungswerkzeuge bieten Wikis jedoch nur eingeschränkte Unterstützung bei der Eingabe von Daten. Außerdem wird eine funktionierende Internetverbindung und ein erreichbarer Wiki-Server für das Abrufen und Bearbeiten der Inhalte notwendig. Eine Gefahr sehen [Starke und Hruschka, 2009] bei der Verwendung von Wikis darin, dass sie ständige Pflege benötigen, damit diese sich nicht zu „chaotischen Datengräbern“ entwickeln.

Zur Beschreibung der Softwarearchitektur werden neben textuellen Erläuterungen Diagramme eingesetzt, welche die Zusammenhänge veranschaulichen und die Übersicht fördern. Diese können in unterschiedlichen Werkzeugen erstellt werden. Beispiele hierfür sind Werkzeuge für die Erstellung von Unified Modelling Language (UML)- und/oder Entity-Relationship (ER)-Diagrammen.

2.3 Fehlerbegriff

Zweck des Testens ist es Fehler zu entdecken [Myers, 1979]. Der Fehlerbegriff ist jedoch mehrdeutig. In Abhängigkeit des Kontext kann darunter die fehlerverursachende Handlung, der Fehler als Teil eines Programms oder dessen Auswirkung verstanden werden. Begeht ein Programmierer bei seiner Tätigkeit eine fehlerhafte Handlung, wird entsprechend von einer Fehlerhandlung (engl. error) gesprochen. Diese hat einen Fehlerzustand (engl. fault) im Programm zur Folge, welcher sich wiederum in einer mehr oder weniger erkennbaren Fehlerwirkung (engl. failure) symptomatisch äußert [IEEE-24765, 2010]. Sofern nicht anders angegeben, wird in dieser Arbeit unter dem Begriff „Fehler“ die Fehlerwirkung verstanden.

2.4 Test- und Testinformationsbegriff

Ähnlich dem Fehler- werden dem Testbegriff mehrere Bedeutungen zuteil. Generell wird in der Literatur zwischen den Begriffen „Test“ und „Testen“ unterschieden. Im Folgenden werden die Definitionen zu beiden Begriffen aus zwei populären Glossaren untersucht.

In [IEEE-829, 2008] wird der „Test“ als Menge von Testfällen (engl. test cases) und/oder den mit ihnen verbundenen Testvorgehensspezifikationen (engl. test procedures) verstanden. Gleichzeitig kann auch deren Durchführung gemeint sein. Das „Testen“ bezeichnet in ihm

eine Aktivität und/oder deren Durchführung, in welcher ein System oder eine Komponente unter bestimmten Bedingungen ausgeführt, die Resultate überwacht und/oder aufgezeichnet und das Testobjekt anhand dieser gegen seine Spezifikation verifiziert wird. Ein Standard, welcher seine Definitionen bzgl. des Tests an diese anlehnt, ist der [IEEE-24765, 2010].

Das International Software Testing Qualifications Board (ISTQB), welches international Software-Tester qualifiziert und zertifiziert, definiert den „Test“ als Menge von Testfällen und beruft sich dabei auf IEEE-829 [ISTQB, 2012]. Der Begriff des „Testen“ wird jedoch viel umfassender als in IEEE-829 definiert. Er steht nicht nur für einzelne Testaktivitäten und deren Durchführung, sondern für einen Prozess für den gesamten Testlebenszyklus. Das Ziel des Prozesses ist es, die spezifizierten Anforderungen zu verifizieren bzw. zu validieren und vorhandene Fehler zu erkennen. Eine ähnliche Definition des „Testen“ findet sich in [BSI, 1998] der British Standards Institution (BSI).

In dieser Arbeit wird je nach Kontext unter „Test“ bzw. „Testen“, entweder der gesamte Prozess oder einzelne Aktivitäten daraus verstanden, ein Programm oder Programmteile (allg. Testobjekt) bei seiner Ausführung systematisch zu prüfen. Ziel dieser Prüfung ist es, Fehler zu entdecken und das Testobjekt gegen seine Anforderungen zu verifizieren. Die Aktivitäten umfassen die Planung, Durchführung und Auswertung von Tests. Die Tests setzen sich dabei aus einer Menge von Testfällen und/oder Testvorgehensspezifikationen zusammen, deren jeweilige Ausführung wiederum als „Test“ bezeichnet wird. Des Weiteren kann der „Test“ als Oberbegriff für alle Teststufen (siehe Kapitel 2.5) im Testprozess (siehe Kapitel 2.6) stehen.

Testinformationen

Alle Artefakte und die in ihnen enthaltenen Informationen, welche während der Durchführung des Testprozesses (siehe Kapitel 2.6) entstehen, werden als Testinformationen betrachtet. Alle Testinformationen, die ein Modul betreffen, können potenziell in dessen Moduldokumentation aufgenommen werden. Welche Testinformationen schlussendlich in welcher Form in der Moduldokumentation aufgenommen werden, ist im Auswahlverfahren der Testinformationen (siehe Kapitel 2.2) zu klären.

Beispiele für Testinformationen eines Moduls sind die für es relevanten Testkonzepte, die Testspezifikationen und die darin enthaltenen Testfälle, die Testprotokolle und die Testberichte. Die genannten und weitere Artefakte werden in Kapitel 2.6 eingeführt und in Kapitel 2.7 detailliert betrachtet.

Abgrenzung zu statischen Prüfungen

Im Allgemeinen ist mit dem Test die Ausführung des Testobjekts verbunden. Daher wird das Testen auch als dynamisch bezeichnet. Dennoch enthalten viele Standardwerke zu den Grundlagen des Softwaretests Kapitel zur statischen Prüfung (z. B.

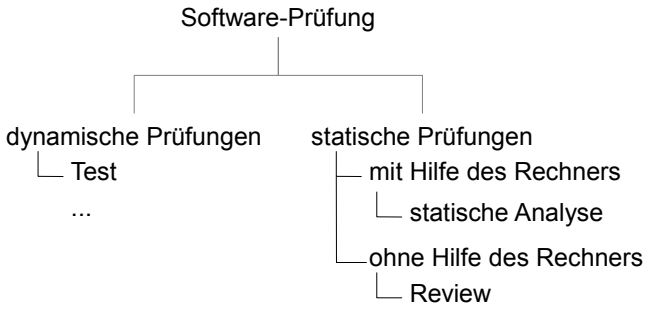


Abbildung 2.1: Klassifikation von Prüfungen in Anlehnung an [Frühauf et al., 2004]

[Liggesmeyer, 2002, Spillner, 2005]). Zu diesen werden Verfahren wie z. B. die statische Analyse oder Reviews gezählt, bei welchen das Testobjekt während der Prüfung nicht ausgeführt wird. Wie auch in [Frühauf et al., 2004] werden sie in dieser Arbeit nicht zum Test gezählt. Die dynamischen und statischen Prüfungen können allgemein als Software-Prüfung bezeichnet werden. Abbildung 2.1 zeigt eine mögliche Klassifikation der Prüfungsverfahren.

2.5 Teststufen

Als Teststufe wird ein Aufwand hinsichtlich des Tests bezeichnet, der über eine eigene Dokumentation und eigene Ressourcen verfügt [IEEE-829, 2008]. Generell wird im Softwaretest zwischen den Stufen Einzeltest, Integrationstest, Systemtest und Abnahmetest differenziert, in welchen das Testobjekt auf unterschiedlichen Abstraktionsebenen getestet wird. In Abhängigkeit des konkreten Projekts können diese Stufen verfeinert oder eigene Stufen – z. B. für Module – definiert werden. Im Kontext von Vorgehens- bzw. Prozessmodellen können die vier generellen Teststufen in entsprechenden Testphasen dargestellt werden.

Abbildung 2.2 zeigt den Softwareentwicklungsprozess nach dem V-Modell von [Boehm, 1979]. Die darin enthaltenen Phasen werden entlang der V-Form durchlaufen. Bei den Phasen der linken Seite handelt es sich um die Entwicklungsphasen. Dem gegenüber stehen die Testphasen auf der rechten Seite. Gegenübergestellt werden solche Phasen, welche sich auf der selben Abstraktionsebene befinden. Charakteristisch für die Abstraktionsebenen ist, dass in der Testphase die Fehler der korrespondierenden Entwicklungsphase am einfachsten aufgedeckt werden können [Spillner, 2005].

Das Ziel der Testphasen ist es, die Resultate der ihnen gegenübergestellten Entwicklungsphase zu verifizieren bzw. zu validieren. Die Validierung bzw. die Verifikation wird durch Testfälle realisiert. Im Folgenden wird kurz auf die genannten Teststufen bzw. -phasen eingegangen.

Einzeltest Der Einzeltest, auch als Komponententest, Modultest oder Unittest bekannt, wird direkt nach der Codierung durchgeführt. In ihm werden die Bestandteile eines Systems isoliert voneinander getestet. Durch die Isolation ist es möglich, Einflüsse durch andere

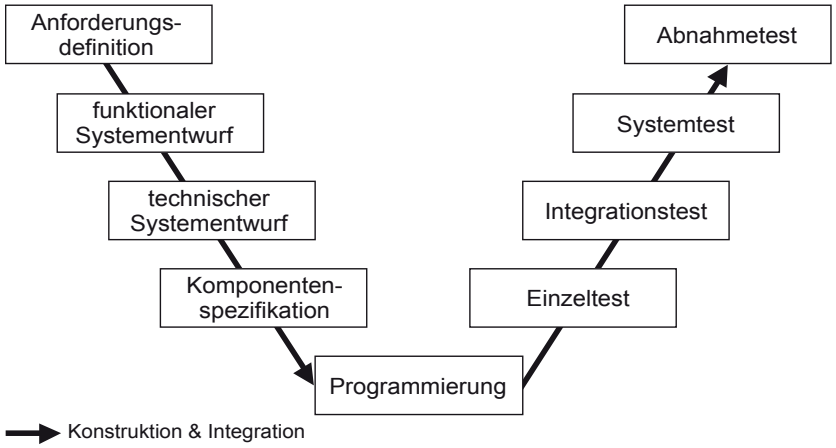


Abbildung 2.2: Das V-Modell nach [Boehm, 1979]

Systembestandteile auszuschließen. Tritt während dem Test eine Fehlerwirkung auf, kann der Suchraum für die Fehlerursache auf das Testobjekt beschränkt werden. Ziel des Einzeltests ist es, das Testobjekt auf die in der Spezifikation festgelegten funktionalen und nicht funktionalen Anforderungen zu testen.

Integrationstest Nach dem abgeschlossenen Einzeltest folgt der Integrationstest. Testobjekte des Integrationstests sind funktional getestete und anschließend zu einem größeren Systembestandteil integrierte Systembestandteil. Ziel des Integrationstests ist es, die Interaktion zwischen den integrierten Systembestandteil zu überprüfen und dadurch semantische und syntaktische Fehler in den Schnittstellen der Systembestandteil ausfindig zumachen.

Systemtest Die dritte Teststufe ist der Systemtest. In ihm werden die spezifizierten funktionalen und nicht funktionalen Anforderungen gegen das entwickelte Gesamtsystem getestet. Das Ziel dieser Teststufe ist es, zu prüfen, ob alle Anforderungen an das Gesamtsystem wie vereinbart umgesetzt wurden. Das System wird dabei in einer Umgebung geprüft, die der Zielumgebung beim Auftraggeber ähnelt oder gleicht.

Abnahmetest Der Abnahmetest wird anders als die vorherigen Teststufen vom Auftraggeber des Entwicklungsprojekts durchgeführt. Die dabei entstehenden Daten unterliegen nicht unbedingt dem Team des Softwareentwicklungsprojekts und werden daher auch nicht von diesem dokumentiert. Entsprechend haben die Testinformationen, welche während dem Abnahmetest entstehen, keine offensichtliche Relevanz für die Moduldokumentation.

Relevanz der Teststufen

Nach den vorangegangenen Abschnitten zu den Teststufen drängt sich die Frage auf, ob die Testinformationen der jeweiligen Teststufe im Auswahlverfahren der Testinformationen

berücksichtigt werden sollen. Da ein Modul nach obiger Definition frei definierbar ist, hängt die Antwort von dem konkreten Modul ab. Folgende Beispielmodule einer objektorientierten Programmiersprache sollen dies verdeutlichen.

Helper_Module besteht aus zwei Klassen, die statische Methoden zum vereinfachten Umgang mit Dateien und Zeichenketten anbietet. Da diese ausschließlich die Standardbibliothek der Programmiersprache verwenden, ist der Einzeltest für den Test des Moduls ausreichend. Entsprechend sollten im Auswahlverfahren der Testinformationen des Moduls die Testinformationen aus dem Einzeltest berücksichtigt werden.

MVC_Module besteht aus drei Klassen, von denen eine das Modell, eine die View und eine den Controller des Model-View-Controller (MVC)-Entwurfsmusters implementiert. Da sich die Klassen untereinander verwenden, wird neben dem Einzeltest ein Integrationstest notwendig. In diesem Fall sollten im Auswahlverfahren der Testinformationen des Moduls neben den Testinformationen des Einzeltests auch die des Integrationstests berücksichtigt werden.

RTA_Module ist eines von vielen Modulen eines größeren Entwicklungsprojekts. Es setzt sich aus mehreren Models, Views und Controllers zusammen, die in ihrer Gesamtheit ein grafisches Rich Text Eingabefeld realisieren. In der Systemspezifikation des Entwicklungsprojekts wurden spezielle Anforderungen für das Rich Text Eingabefeld festgelegt. Dadurch sollten neben den Testinformationen des Einzel- und Integrationstest zusätzlich die des Systemtests beim Auswahlverfahren der Testinformationen des Moduls berücksichtigt werden.

Sofern eine eigene Teststufe für das fragliche Modul definiert wurde, empfiehlt es sich alle Testinformationen dieser Teststufe im Auswahlprozess der Testinformationen des Moduls zu berücksichtigen.

2.6 Der fundamentale Testprozess

Das in Kapitel 2.5 vorgestellte V-Modell suggeriert, dass das Testen erst nach den Entwicklungsphasen, insbesondere der Implementierung, beginnt. Hierbei ist jedoch zu beachten, dass es sich bei den abgebildeten Testphasen lediglich um die Durchführung der Tests handelt. Die Vorbereitung, also die Planung, die Analyse, das Design und die Realisierung der Testfälle, findet bereits parallel zu den Entwicklungsphasen statt. Genauso werden Nachbereitungsarbeiten, wie die Auswertung der Testergebnisse, die anschließende Berichterstattung und der Abschluss der jeweiligen Tests, parallel zu den nachfolgenden Testphasen durchgeführt. Zusammengenommen ergeben die genannten Tätigkeiten ein Vorgehensmodell für den Test, den fundamentalen Testprozess (siehe Abbildung 2.3) [Spillner, 2005, Spillner et al., 2012]. In den Tätigkeiten des fundamentalen Testprozesses werden alle Teststufen berücksichtigt.

Der Testprozess stellt eine Prozessvorlage dar, welche die wichtigsten Tätigkeiten des Testens beinhaltet, diese zeitlich anordnet und in Beziehung zueinander setzt. Wie aber auch bei

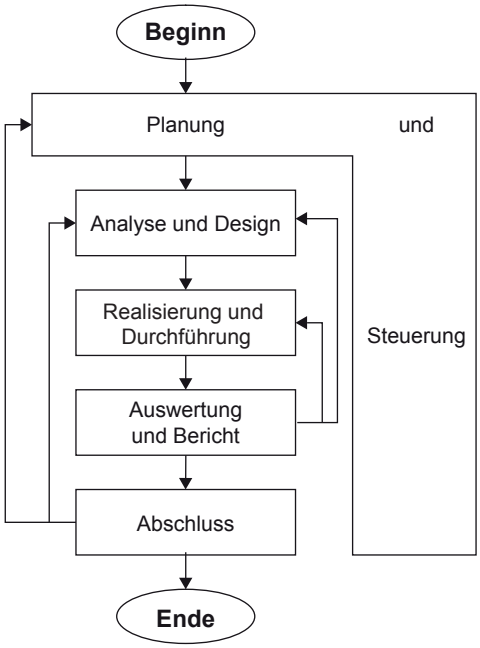


Abbildung 2.3: Der fundamentale Testprozess nach [Spillner et al., 2012]

anderen Vorlagen, ist es sinnvoll ihn an die Rahmenbedingungen des Projekts anzupassen. Dies betrifft sowohl die Phasen, deren zeitliche Abfolge und deren Inhalte.

Neben diesem sehr generischen gibt es weitere Testprozessmodelle, wie z. B. Critical Testing Processes (CTP) [Black, 2003] oder Systematic Test and Evaluation Process (STEP) [Craig, 2002].

Die Gültigkeit der Testprozesse beschränkt sich nicht nur auf das V-Modell, sondern erstreckt sich grundsätzlich über alle Vorgehensmodelle. Diese behandeln den Test durchweg sehr allgemein, was eigenständige Testprozesse notwendig macht.

2.6.1 Planung und Steuerung

Die Planung von umfangreichen und langwierigen Aufgaben ist unumgänglich, sofern der Aufwand, das Budget und die Frist für deren Bewältigung eingehalten werden soll. So auch beim Test.

In der einleitenden Planungsphase wird definiert, welche Ressourcen zu welcher Zeit für welche Dauer benötigt werden und welche Testziele mit welchen -strategien, -verfahren und -werkzeugen erreicht werden sollen. Außerdem wird festgelegt, welche Module mit welcher Intensität getestet werden müssen. Als Grundlage bei der Festlegung sollte das mit den Testobjekten verbundene Risiko für das Gesamtsystem dienen. Die Intensität bestimmt welche Testverfahren jeweils zum Einsatz kommen und bei der Erreichung welcher Kriterien

ein Test als beendet betrachtet werden kann. Diese sogenannten Testendekriterien hängen stark vom eingesetzten Testverfahren ab. So kann beispielsweise gefordert werden, dass alle Testfälle ohne Fehler durchgeführt wurden. Weitere Beispiele für allgemeine Testendekriterien finden sich in [Ludewig und Lichter, 2007]. Neben den Testendekriterien werden des Weiteren Kriterien für den Abbruch bzw. die Unterbrechung und Wiederaufnahme der Tests definiert.

Eng verbunden mit der Bestimmung der Intensität ist die Priorisierung von Tests. Sie sorgt dafür, dass elementare Systemteile zuerst getestet werden. Das spielt spätestens eine Rolle, wenn ein Softwareprojekt in zeitlichen Verzug gerät und dadurch ein ausführlicher Test nicht mehr möglich ist.

Die Steuerung findet parallel zu allen Phasen des Prozesses statt. Sie beinhaltet die Überwachung der Testphasen und den Vergleich von Soll- und Ist-Zustand der Phasen bzw. der darin enthaltenen Tätigkeiten. Des Weiteren werden in ihr Planungsabweichungen dokumentiert und auf diese durch entsprechende Planungsänderungen reagiert.

2.6.2 Analyse und Design

In der Analyse wird die Testbasis und das Testobjekt auf ihre Testbarkeit hin untersucht. Die Testbasis kann abhängig von der Teststufe z. B. die Systemspezifikation, die Grobarchitektur, die Feinarchitektur oder andere Dokumente enthalten. Die Testbarkeit ist gewährleistet, sofern die Testbasis alle Informationen enthält, welche notwendig sind, um die zur Erreichung der Testziele notwendigen Testfälle aus der Testbasis abzuleiten. Abgeleitet wird über die Testverfahren der jeweiligen Teststufe.

Im Design werden zunächst logische Testfälle aus der Testbasis gewonnen, welche in der Realisierungsphase des Testprozesses konkretisiert werden. Logische unterscheiden sich insofern von konkreten Testfällen, da sie für die Ist-Eingaben und Soll-Ausgaben keine Werte definieren, sondern Platzhalter oder Wertebereiche. Neben den Ein- und Ausgaben müssen zu jedem Testfall nach [IEEE-829, 2008] die für die Ausführung notwendigen Vorbedingungen, die erwarteten Nachbedingungen und das Testobjekt spezifiziert werden. Hierbei handelt es sich um die Mindestangaben. In der Praxis besitzen Testfälle zusätzliche Daten, wie einen Status, eine Identifikationsnummer (ID), eine Priorität, einen Autor oder die durch ihn getesteten Anforderungen der Systemspezifikation. Der letzten Information kommt besondere Wichtigkeit zu. In Kombination mit den Testfallprotokollen, die während der Ausführung entstehen, kann durch sie festgestellt werden, welche Anforderungen getestet wurden und ob die mit ihr verbundenen Testläufe Fehler aufgedeckt haben. Diese Rückverfolgbarkeit zwischen Anforderungen und den in der Entwicklung und dem Test entstandenen Artefakten wird allgemein hin als Traceability bezeichnet.

Nachdem die logischen Testfälle dokumentiert wurden, wird abschließend die für deren Ausführung benötigte Testinfrastruktur ermittelt und – soweit möglich – eingerichtet und getestet.

2.6.3 Realisierung und Durchführung

In der Realisierungs- und Durchführungsphase werden zunächst zu den in der vorherigen Phase entstandenen logischen Testfällen konkrete spezifiziert und gruppiert. Die konkreten Ein- und Ausgabewerte der Testfälle werden dabei aus der Spezifikation des Testobjekts gewonnen. Die Gruppierung dient zum einen der Übersichtlichkeit und zum anderen der effizienten Durchführung des Tests. Testfälle können z. B. nach dem Testobjekt oder einer Anforderung in sogenannten Testszenarien logisch gruppiert werden. Das Ziel eines Testszenarios ist es, bestimmte Funktionen des Testobjekts zu testen. Neben den Testsequenzen und anderer Testmittel gibt es die Beziehungen zu anderen Testszenarien an, die vor, nach oder parallel zu ihm ausgeführt werden [IEEE-829, 2008]. Testsequenzen ihrerseits definieren, in welcher Reihenfolge eine Menge von Testfällen ausgeführt werden soll. Entsprechend sind die Nachbedingungen eines Testfalls der Testsequenz die Vorbedingungen seines Nachfolgers. Eine Testsequenz stellt somit eine sequenzielle Abfolge einer Menge von Testfällen dar.

Anschließend werden die Verantwortlichkeiten und die zeitliche Organisation der Tests in der Durchführung festgelegt. Bevor die Testfälle eines Testobjekts ausgeführt werden können, muss die Vorbereitung der korrespondierenden Testinfrastruktur abgeschlossen und das Testobjekt freigegeben sein. Die tatsächliche Durchführung der Tests und die damit verbundene Protokollierung der Testergebnisse schließt die Phase ab.

Das Hauptaugenmerk bei der Testdurchführung liegt auf dem Vergleich der spezifizierten und der tatsächlichen Ausgabe eines Testfalls. Weichen diese voneinander ab, liegt vermutlich ein Fehler vor, welcher in Form einer Abweichungsmeldung festgehalten werden muss. Eine Ausnahme bildet der Fall, in dem die Testspezifikation selbst Fehler enthält. Dies kann zu einem falsch positiven Testresultat führen. Daher sollte die Testspezifikation vor der Testdurchführung einer Prüfung, z. B. in Form eines Reviews, unterzogen werden.

Die Protokollierung der Testergebnisse ist für die Steuerung des Testprozesses und den Nachweis über die Testdurchführung von elementarer Bedeutung. Entsprechend müssen alle Tests minutiös protokolliert werden. Zentraler Bestandteil eines Testfallprotokolls ist die Ist-Ausgabe eines konkreten Testfalls. Daneben sind Informationen zur Ermittlung des Testendes und des tatsächlichen Zeitaufwands festzuhalten. Da sich Testprotokolle auf eine bestimmte Version eines Testfalls beziehen, welcher aus diversen Gründen im Nachhinein angepasst werden könnte, ist die referenzierte Version zu archivieren oder sind die Daten des Testfalls im Protokoll aufzunehmen. Gleiches gilt z. B. für die Version des Testobjekts oder der Testinfrastruktur.

2.6.4 Auswertung und Bericht

In dieser Phase wird ausgewertet, ob alle Testendekriterien der in der vorherigen Phase durchgeführten Tests erfüllt sind. Ist dies nicht der Fall, muss ein weiterer Testlauf ausgeführt und gegebenenfalls die Testspezifikationen der mit den Kriterien verbundenen Tests erweitert oder geändert werden. Auch eine Testplanungsänderung ist denkbar, sofern z. B. mehr bzw.

andere Ressourcen durch die geänderte Spezifikation notwendig geworden sind. Ist ein spezifiziertes Testendekriterien an sich jedoch nicht erreichbar, wird dieses angepasst oder verworfen.

Nachdem die ausgewerteten Daten vorliegen, werden sie in einem Testbericht je Teststufe und einem Gesamttestbericht zusammengefasst. Diese können genauso wie die Protokolle als Grundlage für die Ermittlung des Testfortschritts und der Anpassung der Testplanung im Rahmen der Steuerung dienen.

2.6.5 Abschluss

Zum Abschluss des Testprozesses wird sichergestellt, dass alle vorherigen Phasen tatsächlich abgeschlossen, die darin entstandenen Resultate dokumentiert und archiviert, und an entsprechende Stellen weitergeleitet wurden. Archiviert und weitergeleitet werden nicht nur die in den einzelnen Phasen entstandenen Dokumente, sondern auch die Testmittel. Unter Testmittel werden testrelevante Mittel wie z. B. die Testfälle, die Testprotokolle, die Testinfrastruktur oder die Werkzeuge verstanden. Diese können beispielsweise als Grundlage für die Wartung oder Weiterentwicklung der entstandenen Software und dem damit verbundenen Regressionstest dienen. Ein weiterer Verwendungszweck ist die kontinuierliche Verbesserung des Testprozesses und der Prozessplanung.

2.7 Testdokumentation nach IEEE-829

Im Testprozess entstehen viele Daten. Aus diesen sollten die Wichtigsten ausgewählt und in geeigneter Form dokumentiert werden. Dafür gibt es einige Gründe. Durch die Dokumentation kann der Test bewertet und wiederholt werden. Sie macht diesen insofern bewertbar, da er durch sie selbst einer Prüfung unterzogen werden kann. Das kann entscheidend sein, sofern z. B. vor Gericht nachgewiesen werden muss, dass eine Software mit angemessener Sorgfalt entwickelt wurde. Die Wiederholbarkeit bildet die Grundlage für Regressionstests, die während der Entwicklung aber auch in der Wartung und Weiterentwicklung der Software durchgeführt werden und den Aufwand beim erneuten Testen bestehender Funktionalität reduzieren. Ein weiteres Argument für die Wichtigkeit der Dokumentation des Tests ist, dass Wissen in ihr festgehalten wird, auf welches in nachfolgenden Tests zurückgegriffen werden und zu deren Verbesserung beitragen kann. Nach [Spillner et al., 2012] ist ein nicht dokumentierter Test genauso viel wert wie ein nicht stattgefundener Test.

Neben der Frage, warum die Testdokumentation wichtig ist, stellt sich auch die Frage, welche Daten in welcher Form in ihr festgehalten werden sollen. Richtungsweisend bei der Auswahl und Strukturierung der Daten kann beispielsweise der IEEE-829-Standard sein. In ihm wird eine Menge an Testdokumenten vorgeschlagen. Des Weiteren enthält er Vorschläge bezüglich deren Inhalte und deren Struktur. Sowohl die Dokumenttypen, als auch deren Struktur und Inhalte werden in der Praxis an die Gegebenheiten und Rahmenbedingungen des Projekts angepasst. Zu den vorgeschlagenen Dokumenten gehören das Testkonzept, der Stufentestplan,

das Stufentestdesign, die Stufentestfallspezifikation, die Stufentestvorgehensspezifikation, das Stufentestfallprotokoll, die Abweichungsmeldung, der Stufentestverlaufsbericht und der Stufentestergebnisbericht.

Testkonzept Das Testkonzept (engl. master test plan) enthält die Testplanung des Entwicklungsprojekts und entsteht in der Planungsphase des Testprozesses. Im Wesentlichen besteht es aus zwei Teilen. Ein Teil, der eine Übersicht über den gesamten Test enthält, und einen, der die Details des Testkonzepts erläutert.

Im ersten Teil wird beschrieben, wie der Testprozess mit den anderen Prozessen des Entwicklungsprojekts zusammenspielt, welchen Zeitplan er verfolgt, welche Ressourcen in ihm benötigt werden und welche Methoden, Techniken und Werkzeuge zum Einsatz kommen. Des Weiteren werden die im Test relevanten Metriken, die Verantwortlichkeiten und das Integritätslevelschema (engl. integrity level scheme) festgelegt. Im Integritätslevelschema werden unterschiedliche Integritätslevel definiert. Ein Integritätslevel gibt an, mit welchen Folgen zu rechnen ist, wenn Fehler in der Software, an die das Integritätslevel vergeben wurde, auftreten. Ein höheres Integritätslevel ist im Vergleich zu einem niedrigeren mit einem ausführlicheren Test und einer detaillierteren Testdokumentation der korrespondierenden Software verbunden. Im Standard werden vier Integritätslevel vorgeschlagen, welche an das Gesamtsystem oder Systembestandteile vergeben werden können.

Im zweiten Teil des Testkonzepts wird der eingesetzte Testprozess, die darin enthaltenen Phasen und die Teststufen identifiziert und beschrieben. Außerdem wird festgelegt, welche Testdokumente neben dem Testkonzept existieren und welche Anforderungen an diese gestellt werden.

Stufentestplan In umfangreichen Projekten mit vielen Testteams kann es sinnvoll sein, Pläne für jede der Teststufen anzufertigen. In diesen sogenannten Stufentestpläne (engl. level test plans) werden die zum Teil noch allgemeinen Informationen aus dem Testkonzept hinsichtlich der jeweiligen Teststufe soweit wie möglich konkretisiert. Außerdem wird in ihnen angegeben, welche Testobjekte hinsichtlich welcher Aspekte getestet oder nicht getestet werden sollen. Ein Stufentestplan kann für die in Kapitel 2.5 beschriebenen Teststufen, aber auch für andere erstellt werden. Als Beispiel für weitere Teststufen nennt der IEEE-829-Standard Stufentestpläne für Units und Komponenten. Da Units und Komponenten in dieser Arbeit unter den Modulbegriff fallen, können unter anderem also auch Stufentestpläne für Module erstellt werden. Ebenso wie das Testkonzept sind die Stufentestpläne ein Produkt der Planungsphase des Testprozesses.

Stufentestdesign Genauso wie bei den Testplänen kann für jede Teststufe ein eigenständiges Stufentestdesign (engl. level test design) erstellt werden, welches in der Analyse und Designphase des Testprozesses entsteht. Das Stufentestdesign verfeinert – wie der Name bereits andeutet – die Aspekte der Teststufe in Bezug auf das Design des Tests. In dem Zusammenhang werden die Testfälle für diese Teststufe und die Testszenarien identifiziert und separat aufgelistet. Zu den Einträgen der Auflistungen wird jeweils der eindeutige Name und eine grobe Beschreibung festgehalten. Die detaillierten Testfälle

bzw. Testszenarien finden sich in einer der Stufentestfallspezifikationen (engl. level test case) bzw. einer der Stufentestvorgehensspezifikationen (engl. level test procedure).

Stufentestfallspezifikation Die Stufentestfallspezifikation (engl. level test case) beschreibt einen oder mehrere Testfälle einer Teststufe im Detail. Ein detaillierter Testfall umfasst mindestens alle in Kapitel 2.6.3 beschriebenen Daten. Die Stufentestfallspezifikation und die Stufentestvorgehensspezifikation (engl. level test procedure) sind beide Ergebnisse der Realisierungsphase des Testprozesses.

Stufentestvorgehensspezifikation Die Stufentestvorgehensspezifikation (engl. level test procedure) repräsentiert ein Testszenario einer Teststufe, in dem eine Menge von Testfällen der Stufe ausgeführt wird. Die Stufentestvorgehensspezifikation spezifiziert, in welcher Reihenfolge die entsprechenden Testfälle durchgeführt werden sollen.

Stufentestprotokoll Das Stufentestprotokoll (engl. level test log) protokolliert alle relevanten Daten, die bei der Durchführung von Stufentests entstehen. Zu diesen gehören beispielsweise die in Kapitel 2.6.3 beschriebenen Daten. Außerdem listet es die im Testlauf erstellten Abweichungsmeldungen (engl. anaomaly reports) auf.

Abweichungsmeldung Eine Abweichungsmeldung wird erstellt, sofern die Ist-Resultate eines Testfalls von dessen Soll-Resultaten abweichen. Typischerweise beinhalten die Abweichungsmeldungen einen Titel, eine Beschreibung, die Auswirkung auf das Testobjekt, die Priorität, den Protokollant, das Datum der Protokollierung und den Status der Abweichungsmeldung.

Testberichte Die Testberichte fassen bestimmte Daten des Testprozesses zusammen und bewerten sie anhand der mit ihnen verbundenen Ausgangskriterien (z. B. Testende- und Testabbruchkriterien). Ausgewertet werden dabei beispielsweise die Testprotokolle und die analysierten Abweichungsmeldungen. Der IEEE-829 unterscheidet zwischen dem Stufentestverlaufsbericht (engl. level test interim report), dem Stufentestergebnisbericht (engl. level test report) und dem Gesamttestbericht (engl. master test report). Ein Stufentestverlaufbericht stellt einen Zwischenbericht einer Teststufe dar. Stufentestverlaufsberichte werden zu im Testkonzept geregelten Zeitpunkten erstellt. Neben den oben genannten Inhalten kann dieser Testberichttyp optional eine Empfehlung für das weitere Vorgehen beim Testen auf der Teststufe enthalten. Der Stufentestergebnisbericht wird beim Abschluss einer Teststufe erstellt. Abgeschlossen wird die Auswertungs- und Berichtsphase des Testprozesses mit der Erstellung des Gesamttestberichts. Sowohl der Stufentestergebnisbericht, als auch der Gesamttestberichts beinhalten eine obligatorische Empfehlung für das weitere Testvorgehen bzw. den Testabschluss.

Zwischen den Testdokumenten (z. B. Testvorgehensspezifikation, Testfallspezifikation, etc.) bzw. den durch sie dargestellte Entitäten (z. B. Testszenario, Testfall, etc.) existieren Beziehungen. Diese werden in den Dokumenten als einfache Referenzen mit impliziter Semantik repräsentiert. Abbildung 2.4 zeigt ein Beispiel für eine Menge von Testdokumenten des IEEE-829-Standards und deren Referenzierung untereinander. Wie in diesem Beispiel ersichtlich wird, kann die Referenzstruktur komplex werden. Vor allem zwischen den Testfallspezifikationen, den Testvorgehensspezifikationen, den Testprotokollen und den Abweichungsmeldungen. Eine Testvorgehensspezifikation referenziert eine Menge von Testfallspezifikationen. Das

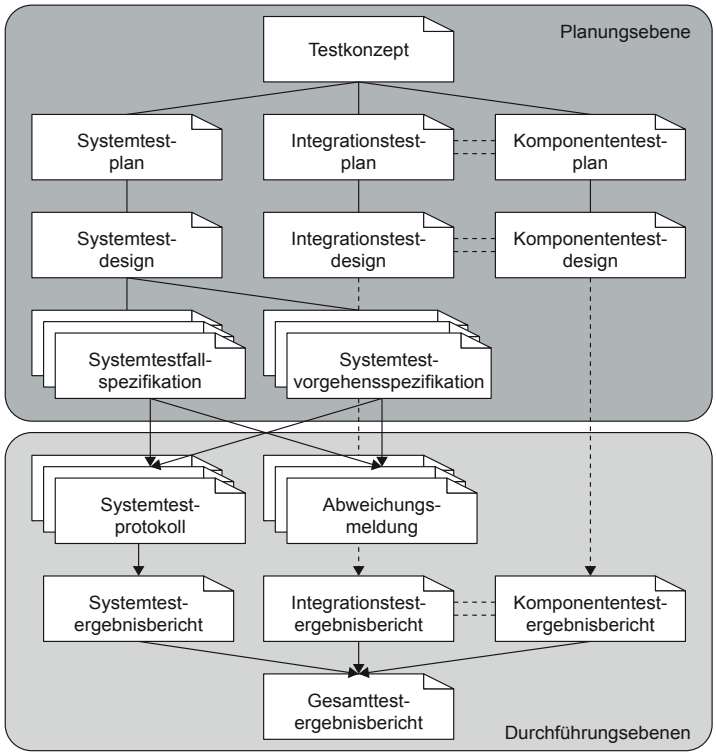


Abbildung 2.4: Beispielhafter Zusammenhang der Testdokumente nach [Spillner et al., 2012]

durch die Testvorgehensspezifikation repräsentierte Testszenario kann in der Realisierungs- und Ausführungsphase des Testprozesses in mehreren Zyklen ausgeführt werden, wodurch pro Ausführung eine Menge von Testprotokollen und gegebenenfalls Abweichungsmeldungen entsteht. Neben den Beziehungen zwischen den Testdokumenten bestehen weitere zwischen diesen und den Anforderungen der Anforderungsspezifikation und den Testobjekten. Die Beziehungen zwischen den genannten Artefakten – also den Testdokumenten, den Testentitäten, den Anforderungen und den Testobjekten – können unterschiedliche Semantiken haben. Beispielsweise kann eine im Testprotokoll enthaltene Referenz auf eine Abweichungsmeldung bedeuten, dass die durch sie identifizierte Abweichung in dem durch das Testprotokoll dokumentierten Testlauf erkannt wurde.

2.8 Werkzeuge zur Unterstützung des Testprozesses

Der Testprozess kann durch die Verwendung von Computer Aided Software Testing (CAST)-Werkzeugen unterstützt werden. In Abhängigkeit der Phase des Testprozesses können unterschiedliche CAST-Werkzeuge – im Weiteren auch Testwerkzeuge genannt – zum Einsatz kommen. [Spillner et al., 2012] unterscheiden zwischen Werkzeugen für das Management und Steuerung von Tests, Testgeneratoren, Analysewerkzeugen, Testdurchführungswerkzeu-

gen und weiteren Werkzeugtypen. Auf letztere wird im Folgenden nicht eingegangen, da sie für die Ausarbeitung keine gesteigerte Relevanz haben. Eine Liste vorwiegend kommerzieller Testwerkzeuge findet sich in [Imbus AG, 2013a]. Einen ausführlichen Katalog zu Open Source Testwerkzeugen bietet [Aberdour, 2013].

Werkzeuge für das Management und Steuerung von Tests Die Planungs- und Steuerungsphase des Testprozesses wird durch diesen Werkzeugtyp unterstützt. Werkzeuge dieses Typs werden auch als Testmanagementwerkzeuge bezeichnet und unterstützen den Benutzer in der Verwaltung von Testfällen. Die Testfälle können in den Werkzeugen erfasst, priorisiert, gruppiert und überwacht werden. Zur Überwachung werden dem Testmanager Funktionen bereitgestellt, um zu verfolgen, ob, wann und wie oft ein Testfall ausgeführt wurde und welche Testresultate mit der Ausführung verbunden sind [Spillner et al., 2012]. Zu diesem Zweck bieten einige Testmanagementwerkzeuge, wie beispielsweise TestBench [Imbus AG, 2013b], Funktionen für die Durchführung der spezifizierten Testfälle.

Des Weiteren stellen viele Werkzeuge Projektmanagementfunktionen zur Verfügung, welche die Zeit- und Ressourcenplanung im Kontext des Testprozesses ermöglichen. Außerdem finden sich in ausgereiften Testmanagementwerkzeugen Funktionen für die Anforderungs-, Fehler- und Konfigurationsverwaltung bzw. Schnittstellen zu darauf spezialisierten Werkzeugen. In ihrer Synergie ermöglicht diese Werkzeugsammlung eine nahtlose Rückverfolgbarkeit zwischen den Anforderungen, den Codeartefakte, den Tests bis hin zu den Fehlermeldungen.

Testmanagementwerkzeuge bieten meist Exportfunktionen für das Berichtswesen an. Durch sie können aus den Daten innerhalb des Testmanagementwerkzeugs einzelne und personalisierte Berichte, bis hin zur gesamten Testdokumentation generiert werden [Spillner et al., 2012].

Für detaillierte Informationen zu den einzelnen auf dem Markt verfügbaren Testmanagementwerkzeugen wird auf die iX-Studie „Software-Testmanagement“ [Illes et al., 2006] verwiesen.

Testgeneratoren Testgeneratoren unterstützen den Testanalysten bei der Erstellung der Testszenarien und der darin enthaltenen Testfälle. Entsprechend kommen sie vor allem in der Analyse- und Designphase des Testprozesses zum Einsatz. [Spillner et al., 2012] unterscheiden zwischen den Testdatengeneratoren und den Testskriptgeneratoren. Erstere werden zur computergestützen Generierung der Ein- und erwarteten Ausgabedaten der Testfälle, letztere zur Generierung des Testablaufs verwendet.

Analysewerkzeuge Mithilfe von Analysewerkzeugen ist es möglich Dokumente, denen eine formale Notation zugrunde liegt, und Code vor und während der Ausführung zu prüfen. [Spillner et al., 2012] unterscheiden zwischen statischen Codeanalysatoren, statische Model Checkern, dynamischen Analysatoren und Überdeckungsanalysatoren. Statische Codeanalysatoren sind jedem Programmierer geläufig. Sie sind in vielen Entwicklungsumgebungen integriert und prüfen den Code auf Merkmale, wie z. B. nicht initialisierte Variablen oder Syntaxfehler. Statische Model Checker vergleichen formal spezifizierte Dokumente gegen das ihnen zugrunde liegende formale Modell.

Ein einfaches Beispiel hierfür ist ein Diagramm, das gegen das Entity-Relationship Modell (ERM) geprüft wird. Der Einsatz von dynamischen Analysatoren ist bei Testobjekten sinnvoll, welche in einer Programmiersprache geschrieben wurden, welche die Speicherverwaltung dem Entwickler überlässt. Sie analysieren während der Laufzeit des Testobjekts Aspekte, wie z. B. dessen Speicherverbrauch und -freigabeverhalten. Überdeckungsanalysatoren ermitteln, welche Codeüberdeckung durch die spezifizierten Tests erreicht wird.

Testdurchführungswerkzeuge Unter dem Begriff „Testwerkzeug“ werden im Allgemeinen Werkzeuge dieses Typs verstanden. Sie unterstützen den Tester, indem sie eine automatisierte Testdurchführung ermöglichen. Bei der Durchführung eines Testfalls versorgt das Werkzeug das Testobjekt mit den Eingabedaten des durchzuführenden Testfalls. Die Ergebnisse des Testobjekts werden vom Werkzeug protokolliert, mit den Soll-Ergebnissen des Testfalls verglichen und erkannte Abweichungen von ihm angezeigt. Für jede der vorgestellten Teststufen existieren unterschiedliche Werkzeuge. Gleiches gilt für Programmiersprachen und für die Art des Tests, wie z. B. dem Graphical User Interface (GUI)-Test. Dementsprechend gibt es viele Werkzeuge für die Unterstützung der Testdurchführung.

Fazit

Für die unterschiedlichen Phasen und Tätigkeiten des Testprozesses existieren spezielle Werkzeuge. Jedes der Werkzeuge konsumiert und produziert bei seinem Einsatz spezifische Daten. Für die Verwaltung dieser Daten werden etablierte Testmanagementwerkzeuge verwendet, die weitreichende Verwaltungs- und Analysefunktionen mitliefern. Die in dieser Arbeit entwickelten Teile von UniMoDoc, sollen keinen Ersatz für die vorgestellten Werkzeugtypen, insbesondere der Testmanagementwerkzeuge, sein. Vielmehr soll sie eine Ergänzung zu ihnen darstellen. In UniMoDoc soll es möglich sein, die Ausgabedaten der CAST-Werkzeuge und die in diesen Werkzeugen verwalteten Daten zu referenzieren. Bei Bedarf soll es dennoch möglich sein, zentrale Testinformationen aus den Ausgabedaten direkt zu verwalten. Die direkte Verwaltung ermöglicht Zeitersparnissen, da die referenzierten Testdokumente nicht geöffnet und auf die gesuchten Testinformationen hin durchsucht werden müssen.

3 Anforderungen, Zielgruppe und Konzept

In diesem Kapitel werden die Anforderungen, die Zielgruppe und das Konzept der in der Diplomarbeit zu entwickelnden Teile von UniMoDoc beschrieben. UniMoDoc ist ein Softwarearchitekturdokumentationswerkzeugs, welches das Ergebnis dieser Diplomarbeit und der von [Pankratz, 2013] ist. Entsprechend stellen die Anforderungen beider Diplomarbeiten zusammengenommen die Anforderungen an UniMoDoc dar. Eine allgemeine Umsetzung der Anforderungen dieser Diplomarbeit wird im konzeptuellen Teil des Kapitels (siehe Kapitel 3.4) beschrieben.

3.1 Funktionale Anforderungen

Nachfolgend werden die funktionale Anforderungen (FA) an die zu entwickelnden Teile aufgelistet:

FA-1 Externe Ressourcen, insbesondere solche, welche Testinformationen darstellen, sollen aus UniMoDoc heraus referenziert werden können.

FA-2 Testinformationen sollen in UniMoDoc direkt verwaltet werden können. Die Verwaltungsfunktionen sind dabei einfach zu halten, da für komplexe Funktionen typischerweise spezialisierte Werkzeuge (siehe Kapitel 2.8) eingesetzt werden.

Zusätzliche funktionale Anforderungen

Im Verlauf der Diplomarbeit hat sich herausgestellt, dass in der Softwarearchitekturdokumentation und zwischen den durch sie referenzierten Testinformationen Beziehungen bestehen, welche – geeignet dargestellt – zur Übersicht in der Modul- und Softwarearchitekturdokumentation beitragen können. Daher ist die Idee aufgekommen, eine Verwaltung und Visualisierung dieser Beziehungen in UniMoDoc zu integrieren. Folgende Anforderungen wurden in dem Zusammenhang identifiziert:

FA-3 Es sollen Beziehungen zwischen den Teilen der Softwarearchitekturdokumentation definiert werden können. Außerdem soll es möglich sein, Beziehungen zwischen den externen Ressourcen zu definieren. Die Beziehungen sollen explizit angelegt, bearbeitet und gelöscht werden können.

FA-4 Beziehungen sollen eine explizite Semantik besitzen.

FA-5 Die Beziehungen sollen in geeigneter Form (z. B. als Graph) visualisiert werden.

FA-6 Die für die Speicherung relevanten Daten sollen in einem textbasierten Format, wie beispielsweise Extensible Markup Language (XML), gespeichert werden.

3.2 Nichtfunktionale Anforderungen

Nachfolgend werden die nichtfunktionale Anforderungen (NFA) an die in dieser Arbeit zu entwickelnden Programmteile von UniMoDoc aufgelistet:

NFA-1 Ebenso wie UniMoDoc sollen sie in der Programmiersprache Java realisiert werden.

NFA-2 Sie sollen gut dokumentiert werden.

NFA-3 Sie sollen einfach erweitert und gewartet werden können.

NFA-4 Die mit den Programmteilen zusammenhängende grafische Oberfläche soll benutzerfreundlich gestaltet sein, so dass eine intuitive Bedienung möglich ist.

3.3 Zielgruppe

Die Zielgruppe der in dieser Arbeit zu entwickelnden Programmteile von UniMoDoc deckt sich mit der Zielgruppe der Softwarearchitekturdokumentation von [Garlan et al., 2010]. Softwarearchitekten können mithilfe von Beziehungen beispielsweise ausdrücken, welche Anforderung durch welche Module bzw. welches Modul realisiert wird oder wie die dokumentierten Module voneinander abhängen. Des Weiteren können die Testinformationen von den Testern eingetragen und/oder referenziert werden. Zusätzlich können die Tester Beziehungen zwischen den referenzierten Testinformationen definieren. In Kombination mit der Visualisierung der Beziehungen können dadurch beispielsweise die Testprotokolle zu einem Testfall einfach ausgemacht werden.

3.4 Konzept

In diesem Kapitel wird das Gesamtkonzept von UniMoDoc vorgestellt. Dieses wurde von [Pankratz, 2013] und dem Autor dieser Arbeit entwickelt.

3.4.1 Documents, Chapters, Sections und Widgets

UniMoDoc ist ein Werkzeug für die Dokumentation von Softwarearchitekturen. Wie in Kapitel 2.2 festgestellt wurde, werden für die Erstellung solcher Dokumentationen meist Textverarbeitungswerkzeuge oder Wikis verwendet. Beide Werkzeugklassen haben gemein, dass mit ihnen Dokumente erstellt werden können. Dokumente werden üblicherweise in Kapitel gegliedert. Entsprechend ist die Zielgruppe der Softwarearchitekturdokumentation – und damit die von UniMoDoc – mit dem Konzept von Dokumenten und Kapiteln vertraut. Um von dem Vorwissen der Zielgruppe profitieren zu können, wird die Softwarearchitekturdokumentation in UniMoDoc auch in Form von Dokumenten und Kapiteln erstellt. Bei den Inhalten von Kapitel 3.4.1 handelt es sich um die konzeptionelle Arbeit von [Pankratz, 2013]. Details zum Konzept finden sich in [Pankratz, 2013].

Documents und Chapters

Ein Dokument wird in UniMoDoc durch ein *Document* dargestellt. Die Kapitel eines Dokuments werden in UniMoDoc als *Chapter* bezeichnet. Ein Document kann sich in eine beliebige Anzahl von Chapters gliedern. Jedes Chapter besitzt genau einen Vater (das Document oder ein anderes Chapter) und eine beliebige Anzahl von Kind-Chapters. Durch diese Rekursion ist es dem Benutzer möglich, eine beliebige Gliederungstiefe im Document zu definieren. Des Weiteren sind Chapters hinsichtlich ihrer Geschwister-Chapters geordnet. In Kombination mit den Vater-Kind-Beziehungen der Chapters ergibt sich daraus eine baumartige Struktur für das Document, in der jedes Chapter eine eindeutige Position besitzt. Die Position eines Chapters spiegelt sich auch in dessen vollständigem Name (z. B. 2.5 Model Test) wider. Dieser setzt sich aus der fortlaufenden und eindeutigen Kapitelnummerierung (z. B. 2.5) und dem benutzerdefinierten Kapitelnamen (z. B. Model Test) zusammen. Mithilfe des vollständigen Namens kann ein Benutzer ein Chapter in einem Document eindeutig identifizieren. Die Position eines Chapters kann bei dessen Erstellung angegeben bzw. im Nachhinein angepasst werden. Sowohl Documents, als auch Chapters werden immer auf der Grundlage eines Templates (siehe Kapitel 3.4.5) erstellt. Sind noch keine benutzerdefinierten Templates vorhanden, kann ein leeres Document bzw. ein leeres Chapter über ein dafür vorgesehenes Template angelegt werden.

Sections und Widgets

Chapters bestehen aus einer beliebigen Anzahl von Abschnitten, welche als *Sections* bezeichnet werden. Jede Section besitzt einen Namen und eine optionale Beschreibung. Außerdem enthält sie die durch den Benutzer eingegebenen Daten der Section und identifiziert ein Widget, mit welchem diese grafisch dargestellt werden sollen. In den Einstellungen der Section kann neben deren Metadaten das identifizierte Widget konfiguriert werden. Dadurch, dass eine Section die Menge von Widgets ausmachen kann, mit welchen ihre Daten darstellbar und verwaltbar sind, kann der Benutzer das Widget in den Einstellungen bei Bedarf wechseln. Ähnlich zu den Chapters des Documents besitzen die Sections eines Chapters eine eindeutige Position in der Struktur des Chapters. Da die Struktur eines Chapters flach ist, bezieht sich die Position einer Section lediglich auf die Reihenfolge im entsprechenden Chapter. Wird eine Section in einem Chapter erstellt, kann ihre Position durch den Benutzer angegeben werden. Ebenso kann der Benutzer diese zu einem späteren Zeitpunkt anpassen.

Wie bereits erwähnt, stellt ein Widget die Daten einer Section grafisch dar und bietet Funktionen an um diese Daten zu bearbeiten. Neben den genannten Aufgaben ist ein Widget außerdem für den Export der durch es verwalteten Daten verantwortlich. Denkbare Formate sind beispielsweise Hypertext Markup Language (HTML) oder Portable Document Format (PDF). UniMoDoc liefert fünf Widgets mit. Für die Darstellung und Bearbeitung von Zeichenketten kann das *TextField*-Widget, das *TextArea*-Widget und das *HtmlEditor*-Widget verwendet werden. Bilder können über das *Image*-Widget eingebunden und Relations (siehe Kapitel 3.4.2) über das *RelationList*-Widget aufgelistet werden.

Darstellung in UniMoDoc

Abbildung 3.1 zeigt UniMoDoc, in dem ein Document geöffnet ist, das mehrere Kapitel enthält. Im linken Bereich, dem Dokumentexplorer (Abbildung 3.1, Nummer:❶), werden die Chapter des Documents in einer Baumstruktur dargestellt. Das Document (Abbildung 3.1, Nummer:❷) ist dabei die Wurzel des Baums und die Chapter die Äste und Blätter. Der Benutzer kann mit den selektierten Elementen des Dokumentexplorers (Abbildung 3.1, Nummer:❸) über das Kontextmenü interagieren. Beispielsweise kann er die Position des Chapters im Document ändern oder es öffnen. Beim Öffnen eines Chapters wird ein neuer Reiter für dieses im mittleren Bereich (Abbildung 3.1, Nummer:❹) erstellt und ausgewählt. In der Abbildung ist Chapter „2.5 Model Test“ in einem Reiter (Abbildung 3.1, Nummer:❺) geöffnet. Der Reiter enthält die Sections des Chapters. In diesem Fall sind das die Sections mit den Namen *Purpose and Responsibility*, *Version*, *Dependencies* und *Testware* (Abbildung 3.1, Nummer:❻). Die Daten der Sections können in den Widgets der Sections (Abbildung 3.1, Nummer:❼) betrachtet und bearbeitet werden. Jeder Reiter verfügt über zwei Modi. Ein Modus, in dem lediglich die Daten der Sections bearbeitbar sind, und einen, in dem neue Sections hinzugefügt, konfiguriert oder gelöscht werden können. Erster wird als Standard-Modus und zweiter als Edit-Modus bezeichnet. Wie der Name bereits andeutet, werden Chapter standardmäßig im Standard-Modus geöffnet. Der Reiter in Abbildung 3.1 befindet sich im Edit-Modus. Über eine Schaltfläche (Abbildung 3.1, Nummer:❽) im

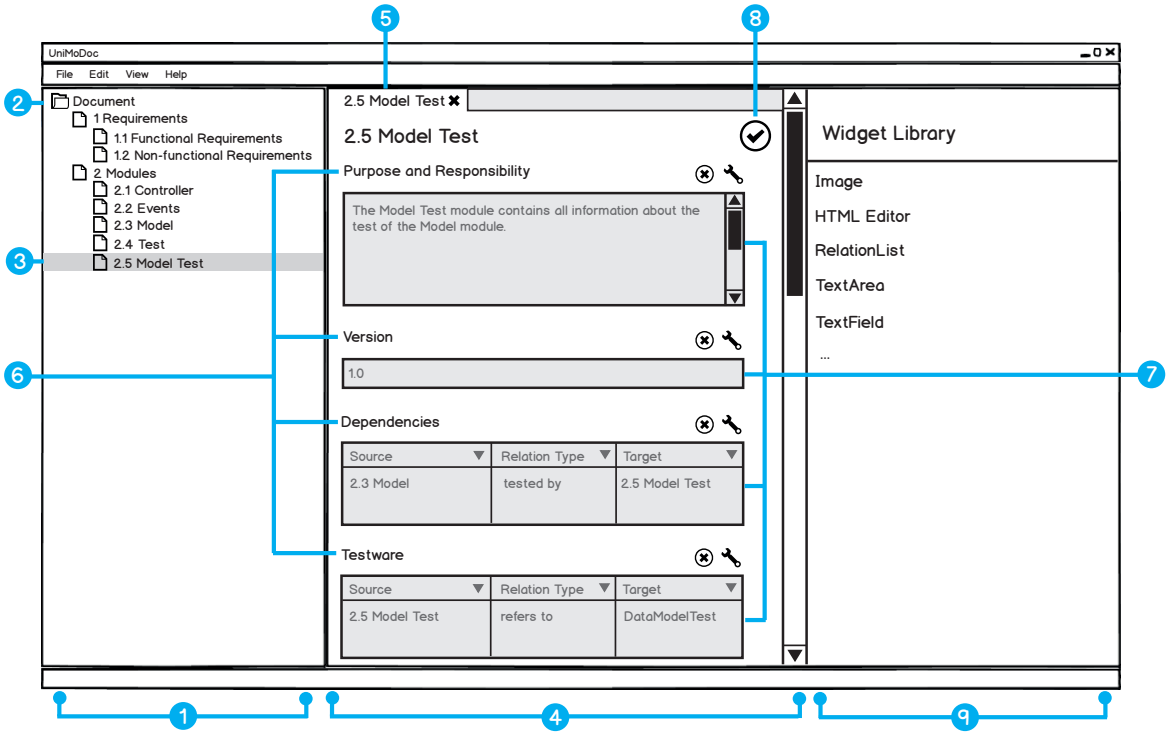


Abbildung 3.1: Mockup von UniMoDoc im Edit-Modus. Bezeichner der Nummern: 1. Dokumentexplorer, 2. Document im Dokumentexplorer, 3. Im Dokumentexplorer selektiertes Chapter, 4. Mittlerer Bereich, 5. Reiter des geöffneten Chapters, 6. Namen der Sections, 7. Widgets der Sections, 8. Schaltfläche um in den Edit-Modus zu wechseln, bzw. diesen zu verlassen, 9. Rechter Bereich

rechten oberen Bereich jedes Reiters kann in den Edit-Modus gewechselt bzw. dieser wieder verlassen werden. Beim Betreten des Edit-Modus wird im rechten Bereich (Abbildung 3.1, Nummer:9) der Anwendung die sogenannte *Widget Library* eingeblendet. Sie zeigt alle in der Anwendung verfügbaren Widgets an. Zieht der Benutzer ein Widget über Drag-and-Drop an eine bestimmte Position des editierten Chapters, wird an dieser eine neue Section mit dem gewählten Widget erstellt.

3.4.2 References, Relations und Relation Types

Um Verlinkungen innerhalb des Documents und zu externen Ressourcen definieren zu können, werden weitere Konzepte notwendig. In Textverarbeitungswerkzeugen und Wikis werden für die Verlinkung einfache textuelle Referenzen (z. B. „siehe Kapitel 2“) und/oder Hyperlinks verwendet.

Diese Art der Verlinkung hat folgende Einschränkungen bzw. Probleme:

1. Die Verlinkungen sind zum einen über das gesamte Dokument verteilt und zum anderen wirken sich Änderungen an einer Verlinkung nicht auf die Verlinkungen aus, welche das gleiche Ziel besitzen. Ändert sich der Standort oder der Name des Ziels, müssen gegebenenfalls alle Verlinkungen gesucht und aktualisiert werden, die das gleiche Ziel verlinken. Da diese über das Dokument verteilt sind, muss das gesamte Dokument durchsucht werden.
2. Sie ist kein geeignetes Mittel, um die Verlinkungen zwischen den externen Ressourcen im Dokument darzustellen. Es kann beispielsweise nicht ausgedrückt werden, dass ein Quellcodeartefakt (wie z. B. eine Klasse) von einem bestimmten Einzeltestfall getestet wird.
3. Die Verlinkungen haben keine explizite Semantik. Der Leser muss daher aus dem Kontext schließen, in welcher Beziehung die Quelle der Verlinkung zu deren Ziel steht. Eine Verlinkung zwischen zwei Kapiteln kann beispielsweise eine in [Garlan et al., 2010] beschriebene *is part of*-Beziehung darstellen. Ist die Verlinkung hingegen explizit mit dieser Semantik gekennzeichnet, kann der Leser feststellen, dass es sich bei den Kapitel um Moduldokumentationen handelt, welche Module repräsentieren, von denen das eine ein Untermodul des anderen ist.
4. Sie ist unidirektional, d. h. die Ziele der Verlinkungen wissen nicht ob und falls ja, von wem sie verlinkt werden. Daher kann der Leser eines Kapitels beispielsweise nicht ohne Weiteres feststellen, welche anderen Kapitel des Dokuments dieses verlinken.

Diese Probleme werden in UniMoDoc mit den *Relations*, *Relation Types* und *References* gelöst. Externe Ressourcen werden in einem Document durch sogenannte References dargestellt. Jede Reference enthält Metainformationen über die durch sie repräsentierte Ressource. Zu den Metainformationen gehört unter anderem der Standort der Ressource (z. B. ein Uniform Resource Locator (URL) oder ein Dateipfad). Ändert sich dieser, muss lediglich die Reference aktualisiert werden (löst Problem 1). Die Relations werden verwendet um Verlinkungen innerhalb des Documents zu definieren. Jede Relation geht von genau einer Quelle aus und zeigt auf genau ein Ziel. Mögliche Quellen und Ziele sind Chapters und References (löst Problem 2). Eine Relation ist immer eine Instanz eines Typs, welcher die Semantik der Relation angibt (löst Problem 3). Diese Typen werden als Relation Types bezeichnet. Alle drei Entitäten sind neben Chapters und Sections Bestandteile eines Documents.

Die Relations, Relation Types und References eines Documents können vom Benutzer im rechten Bereich von UniMoDoc eingesehen werden (siehe Abbildung 3.2). Dieser enthält neben der Visualisierung (Abbildung 3.2, Nummer: ❶) einen Reiter für jede der drei Entitäten. Abbildung 3.2 zeigt den *Relations*-Reiter (Abbildung 3.2, Nummer:❷), in dem alle Relations des Documents in einer Tabelle aufgeführt werden. Jede Zeile der Tabelle entspricht dabei einer Relation. Die Spalten stellen die Eigenschaften der Relations dar. Über die Schaltflächen rechts neben der Tabelle kann der Benutzer neue Relations anlegen und bestehende bearbeiten oder löschen (Abbildung 3.2, Nummer:❸,❹,❺). Neben der Spaltensortierung werden zwei Filter für die Tabelle angeboten. Der eine Filter belässt nur die Relations in der Tabelle, welche in der Visualisierung sichtbar sind. Der andere kann durch den Benutzer konfiguriert werden. In letzterem kann der Benutzer beispielsweise angeben, dass nur die Relations

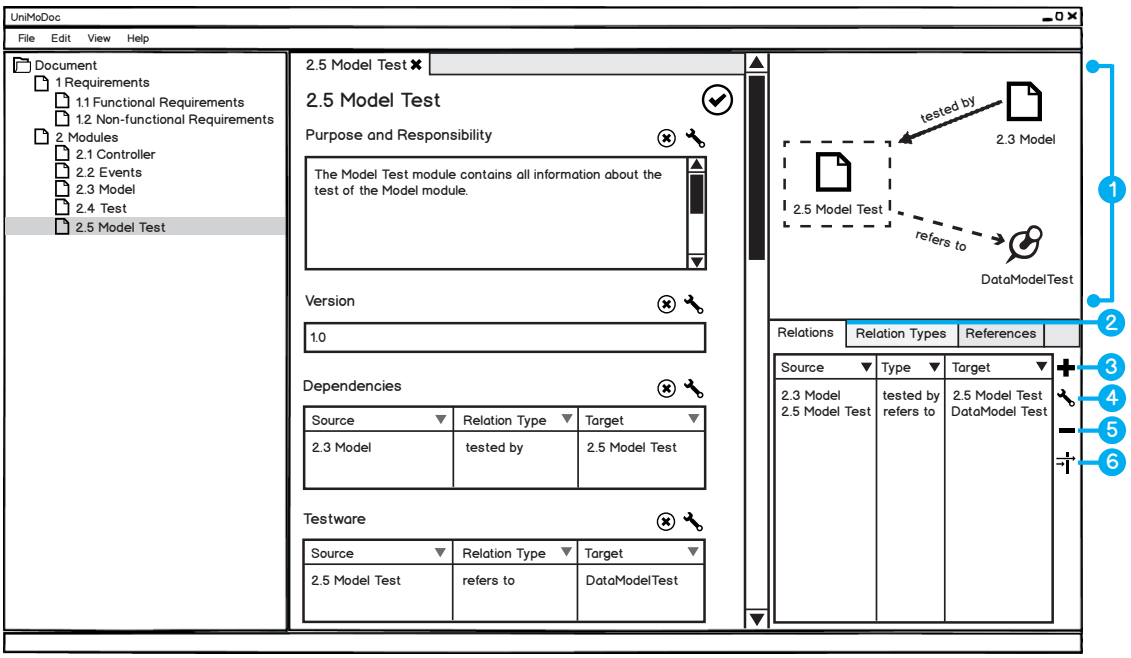


Abbildung 3.2: Mockup von UniMoDoc im Standard-Modus (*Relations*-Reiter). Bezeichner der Nummern:
1. Visualisierung, 2. *Relations*-Reiter, 3. Schaltfläche zum Hinzufügen von Relations, 4. Schaltfläche zum Bearbeiten von Relations, 5. Schaltfläche zum Löschen von Relations, 6. Filtermenü des Reiters

aufgelistet werden sollen, welche ein bestimmtes Chapter als Ziel haben (löst Problem 4). Die Filter können über das Filtermenü (Abbildung 3.2, Nummer:6) ausgewählt und konfiguriert werden. Details zu den Filterfunktionen werden in Kapitel 3.4.4 erläutert.

Der Reiter der References und der Reiter der Relation Types ist analog zum *Relations*-Reiter aufgebaut und bietet dem Benutzer analoge Funktionen für die Verwaltung und Filterung der entsprechenden Entität.

References

Eine Reference repräsentiert eine externe Ressource. Der Standort der Ressource kann entweder über einen Dateipfad, eine URL oder Freitext angegeben werden. Entsprechend ihrem Standort wird eine Reference als lokale Reference, als URL-Reference oder als textuelle Reference bezeichnet. Bei dem Standort handelt es sich um eine Pflichtangabe der Reference. Pflichtangaben sind Eigenschaften, die vom Benutzer angegeben werden müssen, um eine

neue Reference hinzufügen bzw. die Änderungen an einer vorhandenen Reference übernehmen zu können. Analog gibt es auch Pflichtangaben bei Relations und Relation Types. Eine Reference besitzt neben dem Standort außerdem folgende Eigenschaften:

Name Der Name der Reference. Bei dieser Eigenschaft handelt es sich um eine Pflichtangabe. Der Name muss im Dokument einzigartig sein, damit der Benutzer die Reference eindeutig identifizieren kann. Gibt der Benutzer beim Anlegen oder Bearbeiten einer Reference einen Namen ein, welcher bereits an eine andere Reference vergeben wurde, wird er darauf hingewiesen. Die Reference kann erst hinzugefügt bzw. die an ihr vorgenommenen Änderungen können erst übernommen werden, sofern alle Pflichtangaben eingegeben wurden und der Name der Reference einzigartig ist.

Kategorie Die Kategorie der Reference. Durch sie kann der Benutzer References kategorisieren. Beispielsweise kann die Kategorie „Test“ den References zugewiesen werden, welche Testdokumente referenzieren.

Beschreibung Die Beschreibung der Referenz. An dieser Stelle kann der Zweck oder der Inhalt der referenzierten Ressource textuell beschrieben werden.

Autoren Die Autoren der referenzierten Ressource.

Version Die Version der referenzierten Ressource.

Erstellungsdatum Das Erstellungsdatum der referenzierten Ressource.

Änderungsdatum Das letzte Änderungsdatum der referenzierten Ressource.

Kommentare Kommentare zur referenzierten Ressource und/oder Reference.

Besonderheit lokaler References

Dateipfade können unabhängig von UniMoDoc entweder absoluter oder relativer Natur sein. Jede der beiden Alternativen hat gewisse Vor- und Nachteile.

Absolute Dateipfade sind einerseits leicht verständlich, da sie alle Informationen enthalten, um eine Ressource zu identifizieren. Andererseits sind sie nicht für die Serialisierung von Document-Dateien geeignet, da die Pfade die Ressourcen in der Regel auf genau einem Dateisystem identifizieren. Diese Abhängigkeit zu einem bestimmten Dateisystem ist deswegen problematisch für die Serialisierung, weil Document-Dateien für die Versionsverwaltung ausgelegt sein sollen, so dass sie von mehreren Benutzern auf unterschiedlichen Maschinen (mit unterschiedlichen Dateisystemen) geöffnet und bearbeitet werden können.

Relative Dateipfade hingegen sind schwerer zu verstehen als absolute. Zum einen enthalten sie nicht alle Informationen, um eine Datei zu identifizieren, sondern benötigen zusätzlich einen Bezugspunkt, wie z. B. das aktuelle Verzeichnis. Zum anderen ist es notwendig relative Pfade aufzulösen, um den Standort der jeweiligen Ressource zu bestimmen. Dadurch, dass relative Pfade keine Abhängigkeit zum Dateisystem haben, sind sie jedoch bestens für die Versionsverwaltung geeignet.

In UniMoDoc werden die Vorteile beider Pfadtypen vereint. Damit der Benutzer die Standorte von lokalen Ressourcen einfach ausmachen kann, werden ihm deren Dateipfade in der grafischen Oberfläche absolut präsentiert. In Document-Dateien werden die Standorte relativ zum letzten Speicherort des jeweiligen Documents abgelegt. Da der Benutzer eine Document-Datei auf dem Dateisystem potenziell verschieben kann und dadurch die im Document enthaltenen relativen Pfade inkorrekt werden würden, wird der Speicherort des Documents in der Document-Datei selbst festgehalten und von UniMoDoc als Bezugspunkt verwendet, sofern die referenzierten Ressourcen beim Öffnen eines Documents nicht auffindbar sind. Kann eine referenzierte Ressource dennoch nicht gefunden werden, wird der Benutzer von UniMoDoc nach deren Standort auf seinem Dateisystem gefragt. Das kann beispielsweise dann passieren, wenn der Benutzer den Standort oder den Dateinamen einer referenzierten Ressource auf seinem Dateisystem ändert.

Relations

Die Verlinkungen werden in UniMoDoc durch Relations repräsentiert. Eine Relation kann zwischen zwei sogenannten *Relation Endpoint*s definiert werden. Gültige Relation Endpoints sind Chapters und References. Entsprechend kann eine Relation zwischen zwei Chapters, zwischen einem Chapter und einer Reference oder zwischen zwei References definiert werden. Eine Relation besitzt folgende Eigenschaften:

Name Der Name der Relation.

Kommentare Die Kommentare hinsichtlich der Relation.

Quelle Der Relation Endpoint, von dem die Relation ausgeht. Eine Relation besitzt genau eine Quelle. Daher handelt es sich bei dieser Eigenschaft um eine Pflichtangabe.

Ziel Der Relation Endpoint, der das Ziel der Relation darstellt. Eine Relation besitzt genau ein Ziel. Daher handelt es sich bei dieser Eigenschaft um eine Pflichtangabe.

Relation Type Der Relation Type der Relation. Dieser gibt die Semantik der Relation an und bestimmt ihre Darstellung innerhalb der Visualisierung (siehe Kapitel 3.4.3). Eine Relation besitzt genau einen Relation Type. Daher handelt es sich bei dieser Eigenschaft um eine Pflichtangabe. Die Details zu den Relation Types werden weiter unten erläutert.

Relation Types

Der Typ einer Relation wird in UniMoDoc durch einen Relation Type angegeben. Das Konzept der Relation Types wird bereits in anderen Domänen erfolgreich eingesetzt. Ein Beispiel dafür sind die Beziehungstypen von UML. Genauso wie in UML, besitzt jede Relation in UniMoDoc genau einen Relation Type, welcher die Semantik und die Darstellung der Relation vorgibt. Ein Relation Type besitzt folgende Eigenschaften:

Name Der Name des Relation Types. Bei dieser Eigenschaft handelt es sich um eine Pflichtangabe. Der Name muss im Dokument einzigartig sein, damit der Benutzer den Relation Type eindeutig identifizieren kann. Gibt der Benutzer beim Anlegen oder Bearbeiten eines Relation Types einen Namen ein, welcher bereits an einen anderen Relation Type vergeben wurde, wird er darauf hingewiesen. Der Relation Type kann erst hinzugefügt bzw. die an ihr vorgenommenen Änderungen können erst übernommen werden, sofern alle Pflichtangaben eingegeben wurden und der Name des Relation Types einzigartig ist.

Kategorie Die Kategorie des Relation Types. Durch sie kann der Benutzer Relation Types kategorisieren. Beispielsweise kann die Kategorie „Test“ Relation Types zugewiesen werden, welche den Test betreffen.

Beschreibung Die Beschreibung des Relation Types. An dieser Stelle wird angegeben, welche Bedeutung der Relation Type hat. Des Weiteren kann der Relation Type gegen andere abgegrenzt werden. Außerdem kann der Verfasser angeben, zwischen welcher Art von Relation Endpoints oder der durch sie repräsentierten Artefakte, der Relation Type verwendet werden darf.

Darstellungsinformationen Zu jedem Relation Type kann angegeben werden, wie die Relations, welche ihn verwenden, in der Visualisierung grafisch dargestellt werden. Zu diesen Informationen gehören unter anderem die Darstellung des Linienendes, die Linienstärke und deren Farbe. Die Darstellung des Linienendes bestimmt, ob der Relation Type gerichtet oder ungerichtet ist. Linien von gerichteten Relation Types enden mit einem Pfeil, wohingegen ungerichtete ohne Pfeil enden. Die Richtung eines Relation Types hat außerdem Einfluss auf dessen Semantik.

Relation Types der Moduldokumentation

Nachfolgende Relation Types werden für die Moduldokumentation vorgeschlagen. Wird nichts abweichendes zu einem Relation Type angegeben, ist dieser gerichtet.

is part of Der is-part-of Relation Type definiert, dass die Quelle der Relation einen Teil des Ziel darstellt. Die Quelle und das Ziel der Relation stellen Module dar. Dieser Relation Type wird in Anlehnung an die is-part-of Beziehung von [Garlan et al., 2010] definiert.

depends on Mit dem depends-on Relation Type kann eine allgemeine Abhängigkeit zwischen der Quelle einer Relation und deren Ziel ausgedrückt werden. Da es mehrere Formen der Abhängigkeiten gibt, kann dieser Relation Types bei Bedarf in spezifischere aufgeteilt werden. Die Quelle und das Ziel der Relation stellen Module dar. Dieser Relation Type wird in Anlehnung an die depends-on Beziehung von [Garlan et al., 2010] definiert.

is a Der is-a Relation Type gibt an, dass die Quelle einer Relation eine speziellere Ausprägung deren Ziels ist, wobei die Quelle und das Ziel jeweils ein Modul darstellt. Dieser Relation Type wird in Anlehnung an die is-a Beziehung von [Garlan et al., 2010] definiert.

realized by Der realized-by Relation Type definiert, dass die Quelle einer Relation vom Ziel realisiert wird. Da es spezifische Formen der Realisierung gibt, kann dieser Beziehungstype entsprechend verfeinert werden. Durch ihn kann beispielsweise ausgedrückt werden, dass eine Anforderung durch ein Modul oder ein Modul durch eine Menge von Quellcodeartefakten realisiert wird.

tested by Der tested-by Relation Type gibt an, dass die Quelle einer Relation durch deren Ziel getestet wird. Wiederum gibt es mehrere Formen dieses Beziehungstyps. Beispielsweise kann er in einer Relation zwischen einer Anforderung und einem Testfall verwendet werden. Eine weitere Möglichkeit der Verwendung ist die zwischen zwei Modulen, wobei das eine Modul die Testfälle bündelt, welche das andere testen. Ebenso kann der Relation Type zwischen einem Quellcodeartefakt und einem Testfall verwendet werden.

protocolled by Der protocolled-by Relation Type definiert, dass ein Testfall durch ein Testprotokoll protokolliert wird.

uncovers Der uncovers Relation Type gibt an, dass in einem Testprotokoll eine Abweichung zwischen einem Soll- und Ist-Resultat eines Testfalls enthalten ist und erkannt wurde. Diese Abweichung wird im Ziel der Relation in Form einer Abweichungsmeldung festgehalten.

see also See-also ist ein ungerichteter Relation Type. Mit ihm kann ausgedrückt werden, dass sich weiterführende Informationen hinsichtlich der Quelle einer Relation in deren Ziel finden lassen.

refers to Mit dem refers-to Relation Type können einfache Referenzbeziehungen ausgedrückt werden, welche keine tiefer gehende Semantik besitzen.

3.4.3 Visualisierung

Ein Chapter bzw. eine Reference kann potenziell die Quelle bzw. das Ziel von vielen Relations sein. Die Relations können dabei unterschiedliche Relation Types besitzen. Um die Übersicht zu fördern, werden diese Daten im rechten Bereich von UniMoDoc in Form eines Graphen visualisiert.

Die Relation Endpoints, also die Chapters und die References, werden im Graph als Knoten dargestellt. Entsprechend kann ein Knoten entweder vom Typ Chapter oder Reference sein. Damit der Benutzer den Typ eines Knotens auf einen Blick feststellen kann, besitzen diese ein Icon, welches den Typ des Knotens darstellt (Abbildung 3.3, Nummer:❶,❷). Jeder Knoten ist außerdem durch ein Label beschriftet (Abbildung 3.3, Nummer:❸). Als Label dient der Name der durch den Knoten repräsentierten Reference bzw. der vollständige Name des durch ihn repräsentierten Chapters (z. B. 2.5 Model Test). Die Relations werden als Kanten zwischen den Knoten dargestellt (Abbildung 3.3, Nummer:❹). Die Darstellung einer Relation wird von deren Relation Type bestimmt.

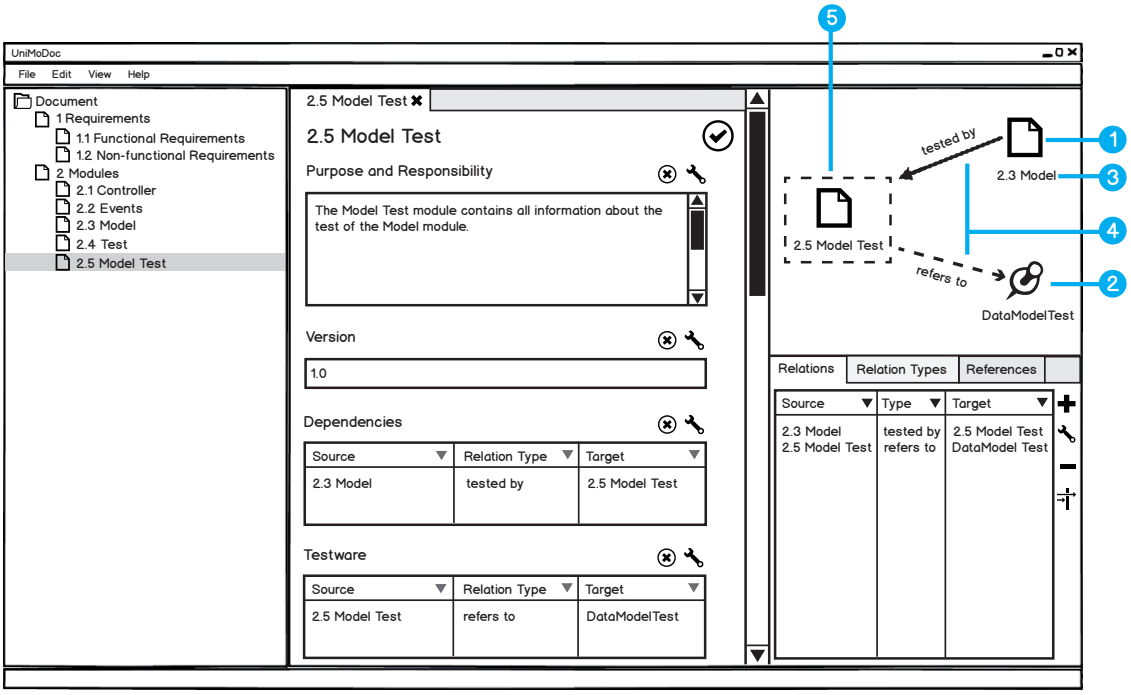


Abbildung 3.3: Mockup von UniMoDoc im Standard-Modus (Visualisierung). Bezeichner der Nummern:
1. Chapter-Knoten, 2. Reference-Knoten, 3. Label eines Knotens, 4. Relations, 5. Fokussierter Relation Endpoint

In der Visualisierung wird immer nur der in ihr *fokussierte* Relation Endpoint (Abbildung 3.3, Nummer:5), seine Relations und die in den Relations vorkommenden Relation Endpoints visualisiert. Standardmäßig wird das geöffnete Chapter (im mittleren Bereich) in der Visualisierung fokussiert.

Entsprechend des oben beschriebenen Konzepts wird nur eine überschaubare Teilmenge der Bestandteile des Documents dargestellt. Eine Darstellung aller Elemente hätte neben potenziellen Performanzproblemen eine schlechtere Übersicht zur Folge. Diese Entwurfsentscheidung ist insofern keine Einschränkung für den Benutzer, da dieser sich vor allem für die direkten Relations eines Chapters bzw. einer Reference interessiert.

3.4.4 Filterung

Die Tabellen des *Relations*-, *Relation Types*- und *References*-Reiters können auf zwei Arten gefiltert werden. Entweder über den *Visualisierungsfilter* oder über den *benutzerdefinierten Filter*.

Der Visualisierungsfilter

Der Visualisierungsfilter ist ein vordefinierter und nicht anpassbarer Filter. Dieser belässt lediglich die Einträge in der entsprechenden Tabelle, welche im Graph visualisiert werden. Ist beispielsweise der *Relations*-Reiter geöffnet und der genannte Filter aktiviert, werden nur die Relations in der Tabelle des Reiters aufgelistet, die zu diesem Zeitpunkt im Graph sichtbar sind.

Der benutzerdefinierte Filter

In jedem Reiter kann ein benutzerdefinierter Filter erstellt bzw. konfiguriert werden. Die Erstellung bzw. Konfiguration des Filters findet über einen Dialog statt. Abbildung 3.4 zeigt den grundlegenden Aufbau des Dialogs, am Beispiel des *Filter Relations*-Dialogs. Der Dialog ist in zwei Bereiche unterteilt. Im oberen Bereich kann der Benutzer eine Eigenschaft der zu filternden Entität – in diesem Fall der Entität Relation – aus einer Combobox (Abbildung 3.4, Nummer:❶) auswählen. Über die Schaltfläche (Abbildung 3.4, Nummer:❷) neben der Combobox, kann eine neue Regel für die ausgewählte Eigenschaft hinzugefügt werden. Alle definierten Regel werden im unteren Bereich des Fensters aufgelistet. Im Beispiel der Abbildung hat der Benutzer zwei Regeln definiert. Eine für die Eigenschaft *Relation Type* (Abbildung 3.4, Nummer:❸) und eine für die Eigenschaft *Name* der Relation. Regeln können vom Benutzer aktiviert bzw. deaktiviert (Abbildung 3.4, Nummer:❹) oder gelöscht (Abbildung 3.4, Nummer:❺) werden. Deaktivierte Regeln werden beim Filtervorgang nicht berücksichtigt, sind jedoch im Filter enthalten. Die Deaktivierung ist dann sinnvoll, wenn eine Regel nur kurzzeitig ausgesetzt werden soll. Hingegen werden gelöschte Regeln vom Filter vollständig und unwiderruflich entfernt. Die Regeln des benutzerdefinierten Filters werden über den logischen *UND*-Operator miteinander verknüpft. Wird also der Filter aus Abbildung 3.4 angewendet, werden in der Tabelle des *Relations*-Reiters nur die Einträge angezeigt, die sowohl den Relation Type „tested by“, als auch den Namen „testRel1“ besitzen.

Einige Eigenschaften der Relations sind im Gegensatz zu denen der References und Relation Types nicht einfach, sondern komplex. Ein Beispiel für eine solche Eigenschaft ist der Relation Type der Relation. Er ist komplex, da er nicht nur aus einem einzigen Wert, sondern aus einer Menge von Werten, nämlich seinen Eigenschaften, besteht. Die Regeln des benutzerdefinierten Filters können nicht nur für die einfachen Eigenschaften, sondern auch für die Eigenschaften der komplexen Eigenschaften definiert werden. Die Eigenschaften der komplexen Eigenschaften werden als transitiv bezeichnet. Durch dieses Konzept kann der Benutzer beispielsweise Relations anhand der Beschreibung ihres Relation Types filtern.

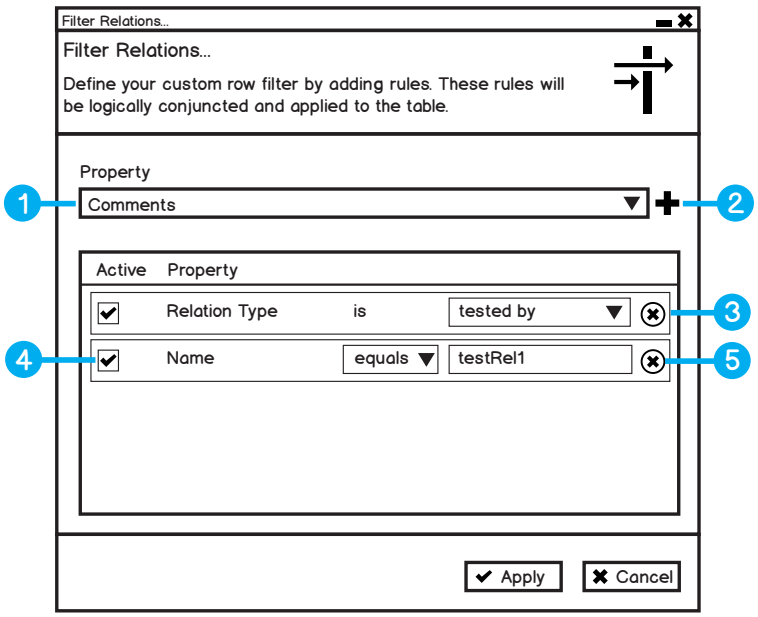


Abbildung 3.4: Mockup des *Filter Relations*-Dialogs. Bezeichner der Nummern: 1. Combobox der Eigenschaften, 2. Schaltfläche zum Hinzufügen einer Regel der gewählten Eigenschaft, 3. Eine Regel, 4. Checkbox zum Aktivieren/Deaktivieren einer Regel, 5. Schaltfläche zum Löschen einer Regel

Wie oben bereits erläutert, enthält die Combobox des Filterdialogs jeweils einen Eintrag für die Eigenschaften der zu filternden Entität. Handelt es sich um eine einfache Eigenschaft, wird lediglich der Name als Bezeichner des Eintrags verwendet (z. B. *Name* oder *Beschreibung*). Für komplexe Eigenschaften wird jeweils ein Eintrag für jede ihrer transitiven Eigenschaften erstellt. Beispielsweise werden für die Eigenschaft Relation Type folgende Einträge im Relations-Filterfenster erstellt:

- Relation Type.Name
- Relation Type.Kategorie
- Relation Type.Beschreibung

3.4.5 Templates

Wie in Kapitel 2.2 erläutert, wird die Softwarearchitektur- und Moduldokumentation meist mithilfe von erprobten Vorlagen erstellt. Durch sie kann Aufwand bei der Erstellung von neuen Dokumenten gespart werden, da keine Zeit für die Strukturierung aufgewendet

werden muss. Außerdem ist durch die Verwendung von Vorlagen sichergestellt, dass für die wichtigsten Inhalte des Dokuments entsprechende Kapitel und Abschnitte vorgesehen sind und diese somit nicht vergessen werden können. Dokumente, die über die gleiche Vorlage erstellt wurden, haben überwiegend dieselbe Struktur. Dadurch können sich die Leser innerhalb der Dokumente schnell zurecht finden, obgleich sie vom Inhalt her eventuell vollständig unterschiedlich sind.

Um von den beschriebenen Vorteilen profitieren zu können, hat [Pankratz, 2013] ein Vorlagenkonzept für UniMoDoc entworfen. Sowohl Documents, als auch Chapters können als Templates abgelegt werden. Jedes Template besitzt einen Namen und eine Beschreibung, über die es identifiziert bzw. über die sein Verwendungszweck beschrieben werden kann.

Legt ein Benutzer ein Document als Template ab, werden alle Daten des Documents, d. h. die Chapter, die Sections und deren Daten, die Relations, die Relation Types und die References, in das Template übernommen.

Wird ein Chapter als Template abgelegt, werden in diesem Template alle Daten des Teilbaums des Documents gespeichert, von welchem das Chapter die Wurzel ist. Somit werden das Chapter, alle direkten und indirekten Kind-Chapters und die in ihnen enthaltenen Sections in das Template übernommen. Ebenso werden diejenigen Relations übernommen, deren Quelle und Ziel jeweils ein Chapter des Teilbaums oder deren Quelle ein Chapter des Teilbaums und deren Ziel eine Reference ist. Außerdem werden die Relation Types und References in das Template aufgenommen, die von den im Template enthaltenen Relations verwendet werden.

Template zur Verwaltung von Testinformationen

Genauso wie in [Garlan et al., 2010] und [Starke und Hruschka, 2009] vorgeschlagen, werden die Testinformationen in UniMoDoc als Teil einer Vorlage der Moduldokumentation verwaltet. Diese Vorlage wird in Form eines UniMoDoc-Templates realisiert, welches in Kapitel 3.4.5 beschrieben wird. Im Folgenden werden die Metadaten erläutert, welche hinsichtlich des Tests in der Vorlage festgehalten werden.

Ansprechpartner Die Ansprechpartner hinsichtlich des Tests des Moduls. Diese Information hilft dem Leser die Ansprechpartner ohne Umstände auszumachen.

Getestete Anforderungen Eine Liste der Anforderungen des Moduls, welche in einem oder mehreren Testfällen getestet werden. Die Resultate der Testläufe sind dabei unerheblich. Es ist lediglich relevant, ob eine Anforderung überhaupt durch einen oder mehrere Testfälle getestet wird.

Kommentare und Sonstiges In diesem Metadatum können die Kommentare hinsichtlich des Tests des Moduls festgehalten werden. Außerdem kann es sonstige Informationen enthalten, die zu keinem anderen Metadatum passen.

Testmittel Eine Liste der Testmittel des Moduls. Hierzu zählen alle referenzierbaren Testinformationen. Beispiele dafür sind der Testplan des Moduls, die Testfälle, die Testprotokolle oder die Testdaten. Dieses Metadatum ist eine Verallgemeinerung der in [Garlan et al., 2010] und [Starke und Hruschka, 2009] beschriebenen Testinformationen.

Vorgehen Beschreibt das geplante und/oder tatsächliche Vorgehen beim Test des Moduls. Die Beschreibung findet typischerweise auf einer abstrakten Ebene statt. Details zum Vorgehen können in den referenzierten Testdokumenten nachgelesen werden. Das Vorgehen beim Test ist auch Teil der Vorlage von [Starke und Hruschka, 2009].

Die aufgeführten Metadaten sind als generische Basis zu verstehen, welche an die Bedürfnisse der Zielgruppe der Softwarearchitekturdokumentation angepasst werden kann. Entsprechend können im Auswahlverfahren der Testinformationen (siehe Kapitel 2.2) weitere Informationen zur Basis hinzugefügt, bzw. bestehende aus ihr entfernt werden. Als Quelle für weitere Metadaten sind die in Kapitel 2.7 aufgeführten und beschriebenen Testdokumente zu betrachten.

Da mit UniMoDoc jegliche Art von Dokumenten erstellt und bearbeitet werden kann, ist es denkbar jedes der Testdokumente des IEEE-829 als eigenständiges Template umzusetzen. Die resultierende Templates könnten anschließend dazu verwendet werden, den gesamten Testprozess eines Softwareentwicklungsprojekts in UniMoDoc zu dokumentieren.

4 Umsetzung

Dieses Kapitel ist der Umsetzung der in Kapitel 3.4 erläuterten Konzepte gewidmet. Eine gute Wartbarkeit und Erweiterbarkeit von UniMoDoc wird durch die Realisierung des MVC-Architekturmusters gefördert. Entsprechend dem Architekturmuster kann jede Klasse von UniMoDoc entweder zum Datenmodell (engl. model), zur Präsentation (engl. view) oder zur Programmsteuerung (engl. controller) gezählt werden. In Kapitel 4.2 wird auf einen Ausschnitt des Datenmodells und in Kapitel 4.3 auf eine wichtige Klasse der Programmsteuerung eingegangen. Anschließend werden noch einige interessante Querschnittthemen behandelt. Zunächst werden jedoch die wichtigsten Technologien aufgezählt, welche bei der Umsetzung von UniMoDoc zum Einsatz kamen. Weitere Informationen zur Umsetzung von UniMoDoc können in [Pankratz, 2013] gefunden werden.

4.1 Eingesetzte Technologien

UniMoDoc wurde unter Verwendung folgender Technologien realisiert:

- *Java 7 Software Development Kit (SDK)*
- *Java Swing*
  Wird für die grafische Benutzeroberfläche verwendet.
- *JAXB*
  Documents und Templates werden in einem textbasierten Datenformat abgelegt. Als Datenformat wurde XML gewählt. Die Serialisierung bzw. die Deserialisierung des Datenmodells von UniMoDoc nach XML bzw. umgekehrt wird über die Java Architecture for XML Binding (JAXB)-Technologie realisiert. Die Gründe für diese Entscheidungen werden in [Pankratz, 2013] erläutert.
- *jGraphX* (v.2.0.0.1)
  Wird zur Visualisierung der Knoten und Kanten eingesetzt.

4.2 Datenmodell

Ein Document besteht aus den in Kapitel 3.4 beschriebenen Chapters, Sections, References, Relations und Relation Types. Documents und Chapters können als Templates für die spätere Verwendung abgelegt werden. Abbildung 4.1 zeigt das Datenmodell dieser Entitäten in einem UML-Klassendiagramm.

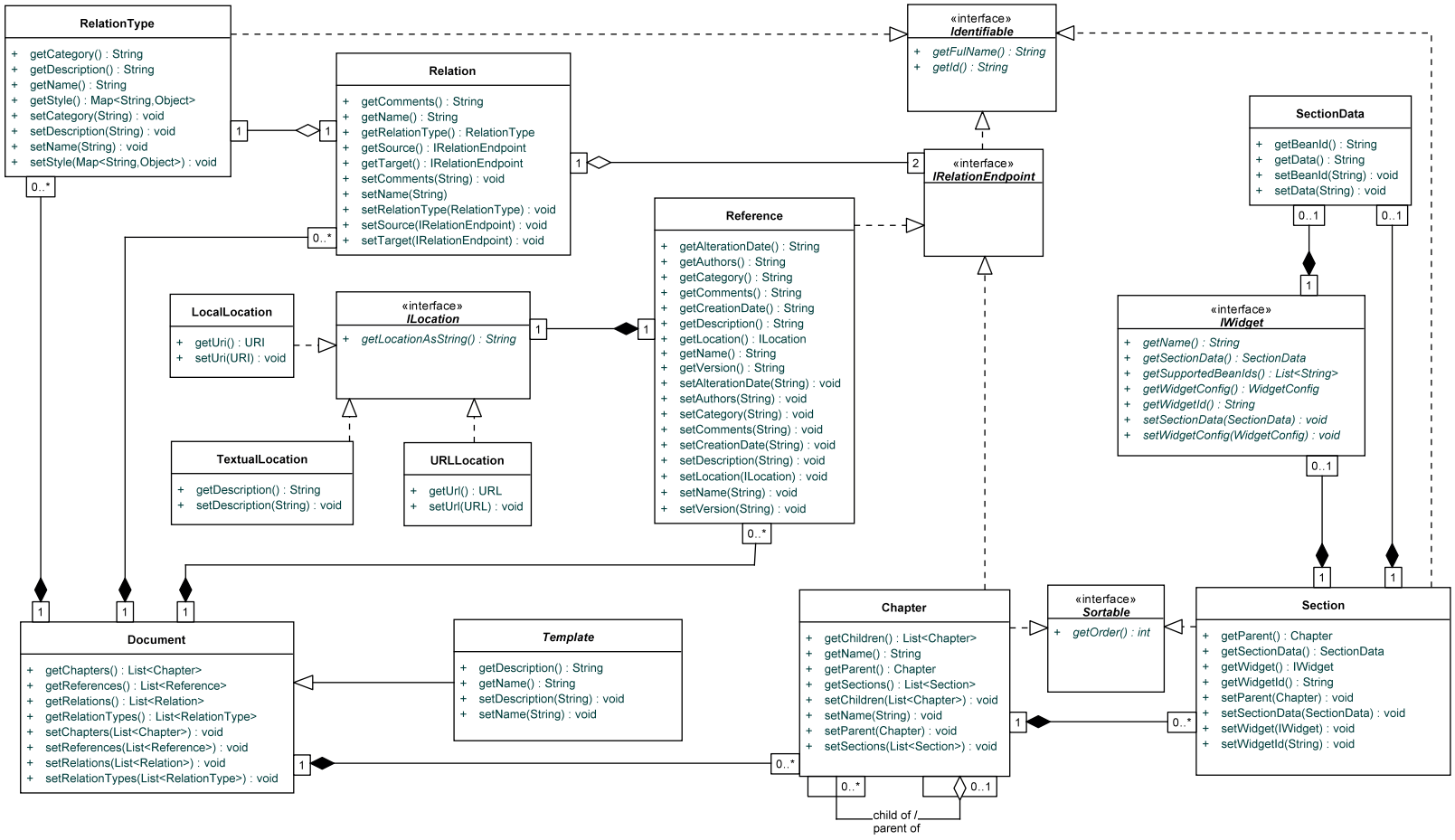


Abbildung 4.1: Klassendiagramm des Datenmodells

Ein Document besitzt jeweils eine Liste von Chapters, References, Relations und Relation Types. Damit die Bestandteile des Documents in der Anwendung eindeutig identifiziert werden können, besitzen sie eine ID. Über die IDs wird sichergestellt, dass die Objektbeziehungen nach dem Speichern und anschließenden Laden eines UniMoDoc-Documents korrekt wiederhergestellt werden können. Außerdem ermöglichen sie die Identifikation eines Objekts, die über die Identifikation via Objektgleichheit hinausgeht. Entsprechend implementieren die Klassen `Relation`, `RelationType` und `Section` die `Identifiable`-Schnittstelle, welche unter anderem eine Methode für die Rückgabe der ID vorsieht (`getId()`). Die Klassen `Chapter` und `Reference` hingegen implementieren die `IRelationEndpoint`-Schnittstelle, welche von `Identifiable` erbt und somit die selben Methodendeklarationen besitzt. Weiter unten wird auf die unterschiedliche Semantik dieser Schnittstellen eingegangen.

`Identifiable` gibt vor, dass es sich bei der ID um einen Wert des Typs String handelt. Die Implementierung der ID-Generierung ist jeder erbenden Klasse selbst überlassen. Alle Bestandteile des Documents verwenden dennoch dasselbe Muster für die IDs:

<Präfix>-<UUID>

Das Präfix ist ein eindeutiges Kürzel der Klasse. Beispielsweise verwendet die Klasse `Relation` das Präfix *rel*. Somit kann über ein Objekt, welches eine ID mit diesem Präfix besitzt, allein anhand dessen ID ausgesagt werden, dass es sich bei ihm um eine Relation handelt. Der Universally Unique Identifier (UUID) wird über die Methode java.util.UUID.randomUUID() erzeugt.

Die Objekte einer Klasse, welche die Schnittstelle `Identifiable` implementiert, besitzen neben `getId()` eine weitere Methode, welche den vollständigen Namen des Objekts als String zurückgibt (`getFullName()`). Dieser wird innerhalb der grafischen Oberfläche dazu verwendet, um das Objekt zu repräsentieren. Anders als die ID kann dieser Wert durch den Benutzer implizit angepasst werden. Beispielsweise kann er ein Chapter umbenennen oder verschieben, wodurch sich dessen vollständiger Name ändert. Die ID des Chapter bleibt von diesen Vorgängen unberührt. Der vollständige Name entspricht bei den Klassen `Relation`, `RelationType` und `Reference` dem Name-Attribut der jeweiligen Klasse.

`IRelationEndpoint` ist eine Markierungsschnittstelle, mit welcher ausgedrückt werden kann, dass die sie implementierende Klasse ein Relation Endpoint darstellt. Entsprechend sind `Chapter` und `Reference` die einzigen Klassen, deren Instanzen die Quelle bzw. das Ziel von Relations sein können. Da es sich bei der Schnittstelle um eine Realisierung des Markierungsschnittstellen-Entwurfsmusters handelt, verfügt sie über keine eigenen Methodendeklarationen.

Die im Konzept vorgestellten Eigenschaften der Relations, Relation Types und References werden in den jeweiligen Klassen als Attribute umgesetzt, welche über entsprechende Methoden abgefragt oder gesetzt werden können. Eine Besonderheit stellen dabei die Darstellungsinformationen der Relation Types und die Standortinformationen der References dar.

Die Darstellungsinformationen werden in einem Relation Type als Schlüssel-Wert-Paare in Form einer `Map<String,Object>` gehalten und können über die Methode `getStyle()`

abgefragt und über die Methode `setStyle()` gesetzt werden. Der Schlüssel der Map ist ein String, der einen eindeutigen Aspekt der Darstellung, wie beispielsweise die Linienfarbe oder die -breite, identifiziert. Die Datentypen der Werte der Map sind abhängig von ihrem jeweiligen Schlüssel. Der Wert der Linienfarbe entspricht beispielsweise dem Datentyp String und der Wert der Linienstärke dem Datentyp Double. Um alle möglichen Datentypen als Wert der Map halten zu können, ist dessen Datentyp Object. Die Darstellungsinformationen werden in der Bibliothek, welche für die Visualisierung verwendet wird (siehe Kapitel 4.5), auf die selbe Art gehalten. Dadurch können unnötige Transformationen zwischen den Darstellungsinformationen der `RelationType`-Klasse und denen der Visualisierungsbibliothek vermieden werden.

Der Standort einer Reference wird im Datenmodell durch eine Implementierung der `ILocation`-Schnittstelle repräsentiert. Entsprechend der drei möglichen Arten von Standorten gibt es die Klasse `LocalLocation` für lokale Dateipfade, die Klasse `UrlLocation` für URLs und die Klasse `TextualLocation` für eine textuelle Beschreibung des Standorts. `ILocation` gibt die Methode `getLocationAsString()` vor, welche gemäß dem Methodenbezeichner den Standort als String-Wert zurückliefert. Dieser wird in der grafischen Oberfläche genutzt, um den Standort einer Reference unabhängig von dessen konkreten Klasse anzuzeigen.

4.3 Programmsteuerung

Eine der zentralen Klassen zur Programmsteuerung ist die Klasse `DocManager`. Jedes `Document`-Objekt ist genau einem `DocManager`-Objekt zugeordnet. Die Aufgabe eines `DocManager`-Objekts liegt darin, das ihm zugeordnete `Document`-Objekt zu verwalten. Die Verwaltung beschränkt sich dabei auf die Manipulation des `Document`-Objekts hinsichtlich dessen direkter Bestandteile. Entsprechend bietet ein `DocManager`-Objekt Methoden an, um Chapter, Relations, Relation Types und References zum ihm zugeordneten `Document`-Objekt hinzuzufügen. Analog hierzu werden Methoden für das Updaten und Löschen dieser Entitäten angeboten.

Die Klassen, deren Objekte an den Änderungen eines `Document`-Objekts interessiert sind, können sich bei dessen `DocManager` über die Methode `addDocumentEventListener` als Listener registrieren. Um sich registrieren zu können, muss die Klasse die Schnittstelle `IDocumentEventListener` implementieren. Die Schnittstelle deklariert Methoden für jede mögliche Änderung an einem Document. Beispielsweise definiert sie solche Methoden, die beim Öffnen oder Schließen eines Documents oder beim Hinzufügen oder Löschen einer Relation ausgeführt werden. In Abhängigkeit der ausgeführten Methode wird dieser ein spezielles Event-Objekt übergeben, welches die Details zur entsprechenden Änderung am Document enthält.

Der prinzipielle Ablauf, der durch eine Änderung an einem `Document` angestoßen wird, soll im Folgenden anhand eines Beispiels verdeutlicht werden. Das Beispiel wird mithilfe des UML-Sequenzdiagramms in Abbildung 4.2 illustriert. Zuerst registriert sich `GraphPanel` *g*, welches für die Visualisierung verantwortlich ist, beim `DocManager` *dm* eines `Documents` *d* (Schritt 1). Anschließend löscht der Benutzer eine `Relation` *r* über die entsprechende

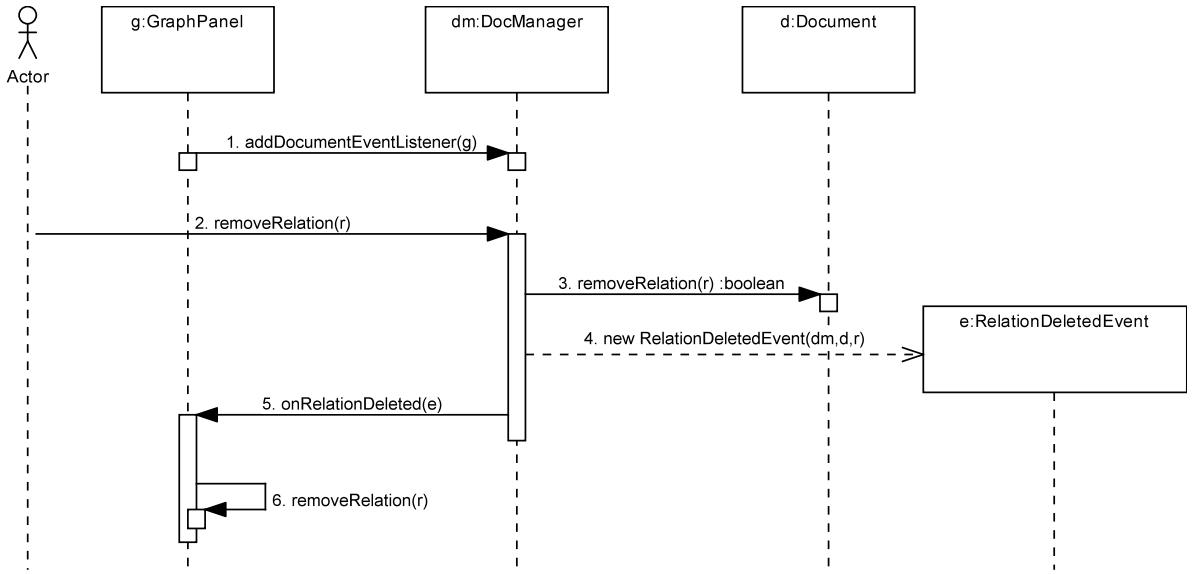


Abbildung 4.2: Ablauf der Propagierung von Änderungen

Schaltfläche im *Relations*-Reiter. Beim Drücken der Schaltfläche wird eine Instanz der `DeleteRelationAction` ausgeführt, die mit der Schaltfläche verknüpft ist. Während der Ausführung wird die Methode `removeRelation()` im `DocManager` *dm* aufgerufen, und dabei die zu löschende Relation *r* übergeben (Schritt 2). *dm* entfernt *r* aus dem `Document` *d* (Schritt 3) und benachrichtigt anschließend die registrierten Listener. Dazu erzeugt er ein neues `RelationDeletedEvent` *e* (Schritt 4). Dem Konstruktor des Events übergibt er neben sich selbst, das `Document` *d* und die Relation *r*. In jedem der Listener wird abschließend die Methode `onRelationDeleted()` aufgerufen (Schritt 5). Dieser wird das erzeugte Event übergeben. Im Beispiel ist nur *g* als Listener in *dm* registriert, weshalb nur *g* benachrichtigt wird. *g* kann abschließend die im Event enthaltenen Informationen dazu nutzen, die Visualisierung gemäß der Änderungen zu aktualisieren (Schritt 6).

4.4 Relations-, Relation Types- und References-Reiter

Die grafische Oberfläche des *Relations*-, *Relation Types*- und *References*-Reiter wird in jeweils einem erweiterten `JPanel`, nämlich dem `RelationsTabPanel`, dem `RelationTypesTabPanel` und dem `ReferencesTabPanel` umgesetzt. Wie bereits im Kapitel 3.4.2 angedeutet, sind diese vom Aufbau her praktisch identisch. Sie unterscheiden sich lediglich in den Tabelleninhalten und den Aktionen, welche mit den Schaltflächen des Reiters verbunden sind. Um die Fehleranfälligkeit zu senken und die Wartbarkeit und Erweiterbarkeit zu verbessern, werden die Gemeinsamkeiten der Panels in einer Oberklasse, `BaseTabPanel`, umgesetzt (siehe Abbildung 4.3). Eine dieser Gemeinsamkeiten ist beispielsweise die Tabelle.

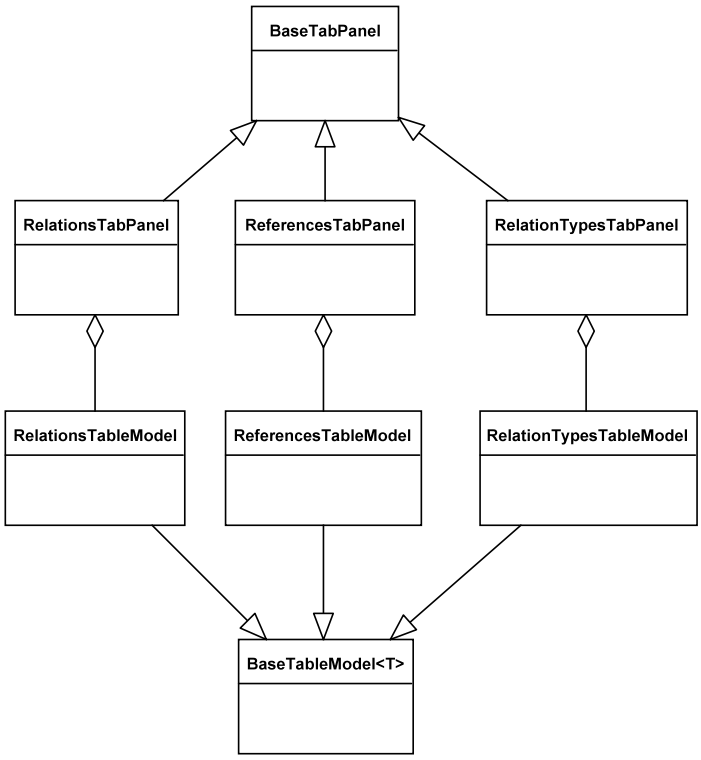


Abbildung 4.3: Zusammenhang der grundlegenden Klassen der Reiter

Tabellen können in Swing durch `JTables` dargestellt werden. Jedes `JTable`-Objekt verwendet ein Objekt einer `TableModel`-Implementierung, um die Daten der Tabelle zu verwalten. Die Methoden, welche die Schnittstelle `TableModel` vorgibt, werden von der Klasse `JTable` verwendet, um die Daten der Tabelle in ihr anzuzeigen. Standardmäßig handelt es sich bei der `TableModel`-Implementierung um die Klasse `DefaultTableModel`. Ein `DefaultTableModel`-Objekt verwaltet die Daten der zugehörigen `JTable` in einer Matrix (ein `Vector` von `Vectors`). Jede Zelle der Matrix entspricht dabei einer Zelle der Tabelle. Die Einträge der Matrix sind vom Datentyp `Object`. `JTable`-Objekte können entweder ein Objekt des standardmäßigen Tabellenmodells oder eines einer anderen Implementierung von `TableModel` als Datenmodell verwenden. Die Verwendung von `DefaultTableModel` hat unter anderem den Nachteil, dass die Daten dem Tabellenmodell, als Array oder `Vector` übergeben werden müssen. Da die Relations, die Relation Types und die References jedoch als Liste im `Document` vorliegen (z. B. `List<Relation>`) und eine Transformation (in einen `Vector` von `Vectors`) umständlich und fehleranfällig ist, wird für die Tabellen der Reiter jeweils ein eigenes Tabellenmodell implementiert.

Analog zu den Panels der Reiter werden die Gemeinsamkeiten der resultierenden Tabellenmodelle, nämlich dem `RelationsTableModel`, dem `RelationTypesTableModel` und dem `ReferencesTableModel`, in der Oberklasse `BaseTableModel` realisiert (siehe Abbildung 4.3). Das `BaseTableModel` implementiert die Methoden der `TableModel`-Schnittstelle und ver-

Listing 4.1 `TableColumn`-Annotation an `getName()`

```
1 private String name;
2
3 @TableColumn(id="relation.name");
4 public String getName(){
5   return this.name;
6 }
```

Listing 4.2 Ein Ausschnitt der Klasse `RelationsTableModel`

```
1 public RelationsTableModel extends BaseTableModel<Relation> implements ... {
2   ...
3   public RelationsTableModel() {
4     super(Relation.class);
5   }
6   ...
7 }
```

waltet die Zeilen einer Tabelle als Liste von Objekten. In Abhängigkeit der Unterklasse von `BaseTableModel` handelt es sich bei den Objekten um `Relations`, `RelationTypes` oder `References`. Die Klassen der drei Entitäten besitzen eine unterschiedliche Anzahl an Attributen. Da die Attribute einer Klasse durch Spalten in der jeweiligen Tabelle repräsentiert werden, haben die Tabellen des *Relations*-, *Relation Types*- und *References*-Reiters wiederum eine unterschiedliche Spaltenanzahl. Damit `BaseTableModel` trotz der Unterschiede mit den Klassen der Entitäten gleichermaßen umgehen kann, werden die Tabellenspalten zur Laufzeit dynamisch erstellt. Der Mechanismus, welcher dabei zum Einsatz kommt, wird im Folgenden anhand eines Beispiels erläutert.

Die Klasse `Relation` besitzt ein Attribut `Name`, welches über eine Methode `getName()` abgerufen werden kann. Diese Methode wird mit der Annotation `TableColumn` versehen. Das `id`-Attribut der Annotation erhält die Zeichenkette *relation.name*. Listing 4.1 zeigt das resultierende Codefragment.

Beim Starten von UniMoDoc wird ein `RelationsTablePanel`-Objekt und darin ein `RelationsTableModel`-Objekt erzeugt. Im Konstruktor des `RelationsTableModel` wird der Konstruktor der Oberklasse `BaseTableModel` aufgerufen (siehe Listing 4.2). Diesem wird die `Class` der in der Tabelle zu verwaltenden Objekte übergeben. In diesem Fall `Relation.class`.

`BaseTableModel` kann nun über die Hilfsmethode `findAnnotatedMethods` die mit `TableColumn` annotierten Methoden der `Class` ausmachen (siehe Listing 4.3). Die Hilfsmethode wiederum verwendet dazu die Reflections API von Java. Die von der Hilfsmethode zurückgelieferte Liste enthält in diesem Beispiel nur einen Eintrag (`getName()`). Jeder Eintrag der Liste repräsentiert eine Spalte der Tabelle im `RelationsTabPanel`. Entsprechend des vorhandenen Eintrags besitzt die Tabelle genau eine Spalte, nämlich die für die Na-

Listing 4.3 Ein Ausschnitt der Klasse `BaseTableModel`

```
1 private List<Method> methods;
2 ...
3 public BaseTableModel(Class<T> clazz) {
4   ...
5   // Find annotated methods of passed class
6   this.methods =
        AnnotationUtils.findAnnotatedMethods(clazz,TableColumn.class);
7   ...
8 }
```

Listing 4.4 Die vereinfachte Methode `getColumnName()` der `BaseTableModel`-Klasse

```
1 public String getColumnName(int column) {
2   Method m = methods.get(column);
3   TableColumn f = m.getAnnotation(TableColumn.class);
4   return Messages.getString(m.getAnnotation(TableColumn.class).id());
5 }
```

men der Relations. Der Name der Spalte und die Werte derer Zellen können über die Methode der Spalte bestimmt werden. Ersterer wird von der Tabelle über die Methode `getColumnName()` abgerufen, welche den Namen mithilfe der `id` der Annotation (*relation.id*) aus einer Properties-Datei ließt und an die Tabelle zurückliefert (siehe Listing 4.4).

Der Wert einer Spalte wird von einer Tabelle über die Methode `getValueAt()` des `TableModels` erfragt. Um beispielsweise den Wert einer Zelle 01 (Spalte 0, Zeile 1) zu erhalten, wird in ihr die Methode der Spalte 0 (z. B. `getName()`) auf dem durch die Zeile 1 repräsentierten Objekt – in diesem Fall einer Relation – ausgeführt. Listing 4.5 zeigt den Code der Methode `getValueAt()`.

Listing 4.5 Die vereinfachte Methode `getValueAt()` der `BaseTableModel`-Klasse

```
1  public Object getValueAt(int row, int column) {
2    T object = objects.get(row);
3    Method m = methods.get(column);
4    TableColumn f = m.getAnnotation(TableColumn.class);
5    try {
6      obj = m.invoke(object, new Object[0]);
7    } catch (Exception e) {
8      logger.error(e);
9    }
10   return obj;
11 }
```

Am obigen Beispiel wurde exemplarisch gezeigt, dass mithilfe der Annotationen Tabellenspalten zur Laufzeit erstellt werden können. Wie oben bereits angeführt, kann dadurch mit den unterschiedlichen Datentypen der Tabellenzeilen auf gleiche Weise umgegangen werden. Ein weiterer Vorteil der dynamischen Erstellung ist der, dass die grafische Oberfläche bezüglich der Tabellen sehr einfach erweitert werden kann. Kommt ein Attribut zu einer Relation, einem Relation Type oder einer Reference hinzu, muss dessen Getter lediglich annotiert und die Properties-Datei um einen Eintrag für den Spaltennamen ergänzt werden.

4.5 Visualisierung

Die Visualisierung von Graphen ist ein sehr zeitaufwändiges Unterfangen. Daher empfiehlt es sich, eine Bibliothek zu verwenden, welche grundlegende Visualisierungsfunktionen anbieten. Zu diesen Funktionen zählt beispielsweise das Zeichnen von Knoten und Kanten in einer mitgelieferten grafischen Komponente.

Als mögliche Kandidaten kamen hierfür nur solche Bibliotheken in Frage, welche...

- ... in der Programmiersprache Java realisiert sind.
- ... eine Darstellungskomponente besitzen, welche mit Swing kompatibel ist.
- ... Open Source und kostenlos sind.

Folgende Bibliotheken haben die Kriterien erfüllt und wurden daher genauer untersucht: jGraphX [JGraph, 2013a], jGraphT [Naveh, 2013] und Java Universal Network/Graph Framework (JUNG) [JUNG, 2013]. Nach Aussage der projekteigenen Homepage ist jGraphT eine Bibliothek, in welcher der Fokus auf Datenstrukturen und Algorithmen liegt. Da diese Aspekte nicht relevant für die Visualisierung in dieser Arbeit sind, schied jGraphT vorzeitig aus. Sowohl jGraphX (v.2.0.0.1), als auch JUNG (v.2.0.1) wurden genauer untersucht. Da JUNG bereits mit der Darstellung einfacher Graphen Schwierigkeiten hatte, die jGraphX hingegen problemlos darstellen konnte, wurde letztendlich jGraphX ausgewählt. jGraphX schien des Weiteren gut für die Umsetzung geeignet zu sein, da die Bibliothek weit verbreitet ist, ihr Datenmodell der Knoten und Kanten vergleichsweise einfach ist und sie über Layoutalgorithmen verfügt. Letztere können dazu verwendet werden, um die Knoten und Kanten des Graphen in der visualisierenden Komponente automatisch anzuordnen.

Dennoch gestaltete sich die Arbeit mit jGraphX schwierig, da die Dokumentation der Bibliothek zum Zeitpunkt der Entwicklung sehr spärlich war. Bei tiefergehenden Fragen mussten meist mehrere Beiträge des jGraph Forums [JGraph, 2013b] zu rate gezogen werden. Diese lieferten in vielen Fällen hilfreiche Hinweise.

Listing 4.6 Die Methode `include()` der `VisualizationRowFiler`-Klasse

```
public boolean include(RowFilter.Entry<? extends TableModel, ? extends
    Integer> entry) {

  // Retrieve GraphPanel
  GraphPanel panel = ObjectRegistry.getInstance().
    getObjectAs(GraphPanel.class,ObjectRegistry.GRAPH_PANEL);

  // Retrieve displayed Identifiables from GraphPanel
  List<String> elementIds = panel.getDisplayedIdentifiableIds();
  if ((entry.getModel() instanceof IFilterableTableModel)) {
    IFilterableTableModel model = (IFilterableTableModel) entry.getModel();
    IFilterableRow row = model.get(entry.getIdentifier());
    return elementIds.contains(row.getId());
  }
  return false;
}
```

4.6 Filterung

`JTables` können in Java über einen Zeilenfilter gefiltert werden. Jeder Zeilenfilter realisiert die abstrakte Klasse `RowFilter`. Diese enthält die abstrakte Methode `include()`, welche auf jede Zeile der Tabelle angewendet wird, sobald die Tabelle den Filter verwendet. Liefert die Methode den Wahrheitswert `true` zurück, wird die Zeile in der Tabelle angezeigt, andernfalls wird sie es nicht. Gemäß dieser Praxis, werden der Visualisierungsfilter und der benutzerdefinierte Filter als `RowFilter` implementiert.

Der Visualisierungsfilter wird in der Klasse `VisualizationRowFiler` umgesetzt. Diese ist so allgemein gehalten, dass sie für jede der Tabellen der drei Reiter verwendet werden kann. In ihrer `include()`-Methode werden zunächst die IDs der im `GraphPanel` derzeit sichtbaren Relations, Relation Types und References vom `GraphPanel` erfragt. Die IDs werden von diesem als Liste zurückgegeben. Anschließend wird geprüft, ob die ID des Objekts, welches durch die zu prüfende Tabellenzeile repräsentiert wird, in der Liste enthalten ist. Falls ja, wird die Zeile angezeigt, andernfalls wird sie ausgeblendet. Listing 4.6 zeigt die Methode.

Da der benutzerdefinierte Filter zur Laufzeit konfiguriert wird, ist eine harte Codierung der `include()`-Methode, wie beispielsweise in der `VisualizationRowFiler`-Klasse, nicht möglich.

Für jede Regel, die in einem Filterdialog durch den Benutzer angelegt wurde, wird zur Laufzeit ein `RowFilter` erstellt. Sobald der Benutzer den Filterdialog bestätigt, werden die `RowFilter` über die Methode `RowFilter.andFilter()`, welche eine Liste von `RowFilters` erwartet, zu einem einzelnen Filter verknüpft. Entsprechend dem Namen der Methode, verknüpft sie die Filter über den logischen *UND*-Operator.

5 Ergebnisse

In diesem Kapitel werden die Ergebnisse der Umsetzung präsentiert. Zunächst wird auf das Template zur Verwaltung von Testinformationen eingegangen, dessen Metadaten in Kapitel 3.4.5 beschrieben wurden. Anschließend wird auf die Reiter der drei Entitäten Relation, Relation Type und Reference und die Visualisierung eingegangen. Abschließend wird an einem Beispiel gezeigt, wie eine neue Relation, ein Relation Type und eine Reference in UniMoDoc angelegt werden kann.

5.1 Template zur Verwaltung von Testinformationen

In Kapitel 3.4.5 wurden Metadaten vorgeschlagen, die in einer Moduldokumentation bezüglich des Tests festgehalten werden können. Diese Metadaten bilden die Grundlage für ein Template, das im Rahmen dieser Arbeit erstellt wurde. Mithilfe des Templates können die Testinformationen eines Moduls in UniMoDoc verwaltet werden.

Abbildung 5.1 zeigt die Dokumentation eines Moduls mit dem Namen *Widgets*. Die Dokumentation enthält unter anderem ein Chapter für die Testinformationen des Moduls: das *Test*-Chapter. Dieses wurde über das oben genannte Template erstellt. Für jedes vorgeschlagene Metadatum enthält das Chapter jeweils eine Section. Die Ansprechpartner (engl. persons in charge), das Vorgehen (engl. approach) und die Kommentare und sonstige Informationen (eng. comments and miscellaneous) bezüglich des Tests des Moduls können in den entsprechenden Sections als Freitext dokumentiert werden. Da sich das *TextArea*-Widget für die Darstellung langer Texte oder einfacher Listen eignet, wird es für diese Sections verwendet. Die Sections des Metadaten *Testmittel* (engl. testware) und *getestete Anforderungen* (engl. tested requirements), verwenden das *RelationList*-Widget. Der Grund dafür ist der, dass in ihnen die Referenzen zu Testmitteln bzw. zu Anforderungen aufgeführt und Referenzen in UniMoDoc durch Relations repräsentiert werden.

Zu den Ergebnissen von [Pankratz, 2013] zählt unter anderem ein Template, mit dem Moduldokumentationen erstellt werden können. Um darin die Testinformationen des jeweiligen Moduls verwalten zu können, verwendet es das dem *Test*-Chapter zugrundeliegenden Template. Im Template von [Pankratz, 2013] wird die Struktur und die Inhalte der in Abbildung 5.1 zu sehenden Chapter *Implementation*, *Design* und *Misc* vorgegeben. Gleiches gilt für deren Vater-Chapter, welches zentrale Informationen eines Moduls beherbergt.

Durch die Kombination beider Templates ist es möglich, Moduldokumentationen in UniMoDoc zu erstellen, in denen die grundlegenden Informationen von Modulen festgehalten werden können.

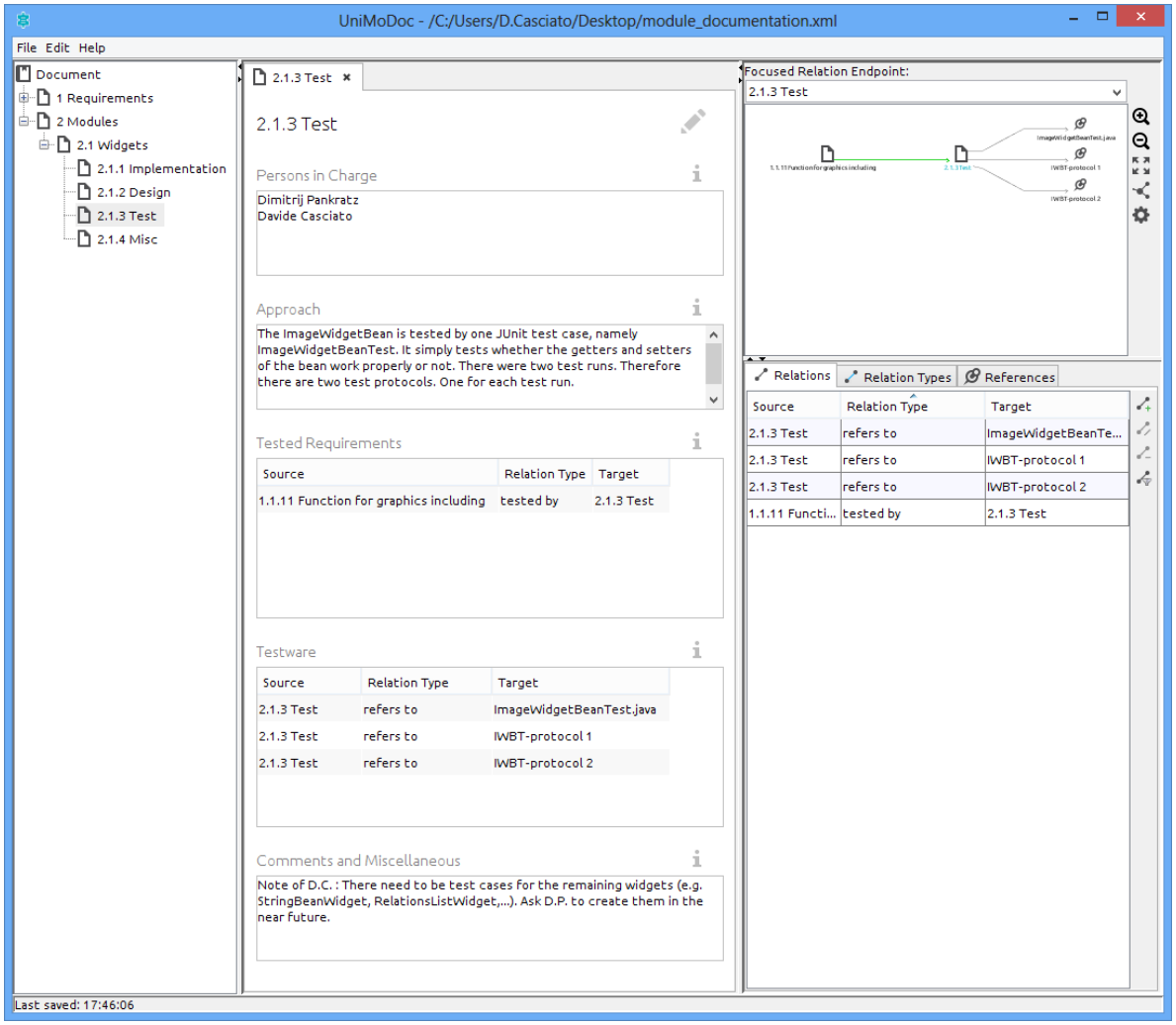
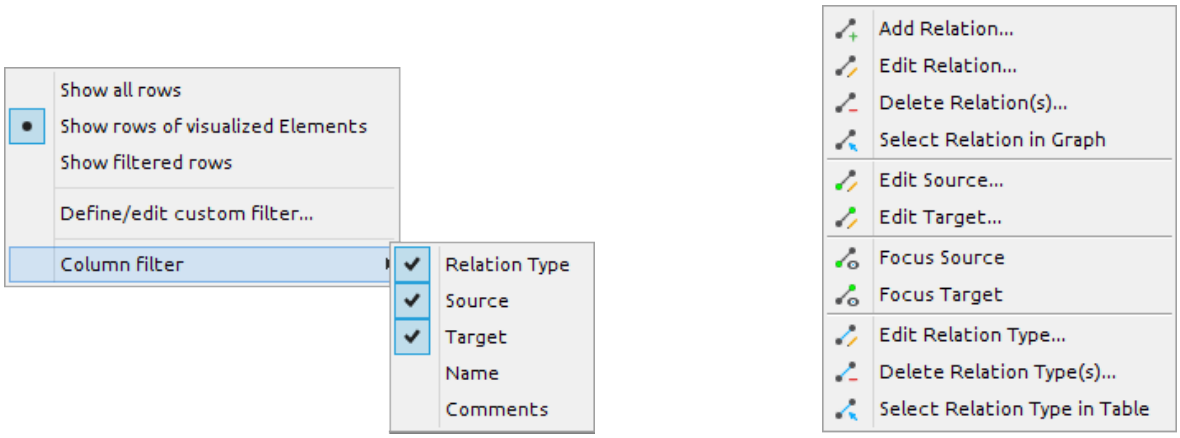


Abbildung 5.1: Moduldokumentation des Widgets-Moduls

5.2 Relations-, Relation Types- und References-Reiter

Die Relations, Relation Types und References eines Documents können über entsprechende Reiter im rechten Bereich von UniMoDoc verwaltet werden. Wie bereits mehrfach erwähnt, sind die Reiter vom Aufbau her identisch und bieten dem Benutzer dieselben Interaktionsmöglichkeiten in Abhängigkeit der jeweils durch sie verwalteten Entität. Im Folgenden werden die Interaktionsmöglichkeiten am Beispiel des *Relations*-Reiters erläutert.

Der *Relations*-Reiter ist in Abbildung 5.1 zu sehen. In ihm ist der Visualisierungsfilter aktiviert. Entsprechend werden in der Tabelle alle Relations aufgelistet, die das fokussierte Chapter als Quelle oder Ziel besitzen. Da in der Visualisierung standardmäßig das aktuell geöffnete Chapter fokussiert wird, handelt es sich bei diesem um das *Test*-Chapter. Möchte



(a) Filtermenü (b) Kontextmenü

Abbildung 5.2: Filter- und Kontextmenü des *Relations*-Reiters

der Benutzer alle Relationen des Documents in der Tabelle sehen, wählt er den Eintrag *Show all rows* des Filtermenüs (siehe Abbildung 5.2(a)) aus. Das Menü kann über die -Schaltfläche neben der Tabelle geöffnet werden. Der benutzerdefinierte Filter lässt sich über den Eintrag *Show filtered rows* aktivieren und über den Eintrag *Define/Configure custom Filter* definieren bzw. konfigurieren. Wählt der Benutzer letzteren Eintrag aus, öffnet sich der Filterdialog des Reiters. Details zum Aufbau und den Inhalten des Dialogs können in Kapitel 3.4.4 nachgelesen werden. Neben den Zeilen können im Filtermenü zusätzlich die Spalten der Tabelle gefiltert werden. Um die Benutzerfreundlichkeit zu verbessern, werden alle vorgenommenen Filtereinstellungen persistent gespeichert.

Mit den Relations des *Relations*-Reiters kann der Benutzer auf zwei Arten interagieren. Entweder über die Schaltflächen neben der Tabelle oder über das Kontextmenü der Tabelle (siehe Abbildung 5.2(b)). Letzteres bietet Interaktionsmöglichkeiten, die über das Bearbeiten, Löschen und Hinzufügen von Relations hinausgeht. Beispielsweise enthält es Einträge, um mit den Relation Endpoints der in der Tabelle selektierten Relation zu interagieren. Diese können über jeweils einen Eintrag bearbeitet oder in der Visualisierung fokussiert werden. Des Weiteren ist eine Interaktion mit dem Relation Type der Relation über entsprechende Einträge des Menüs möglich.

Beim Verweilen des Mauszeigers auf einem Eintrag der Tabelle wird ein Tooltip angezeigt, welcher die Details des jeweiligen Eintrags anzeigt. Durch die Tooltips kann sich der Benutzer die Detailinformationen zum entsprechenden Eintrag sehr einfach und schnell anzeigen lassen.

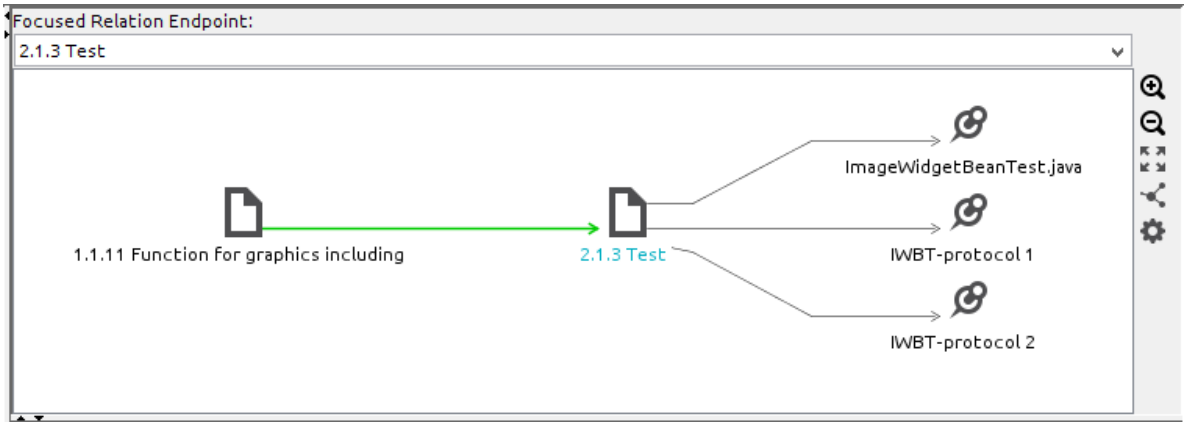


Abbildung 5.3: Ausschnitt der Visualisierung

5.3 Visualisierung

Abbildung 5.3 zeigt die Visualisierung, in welcher das *Test*-Chapter aus Abbildung 5.1 fokussiert ist. Das Label des fokussierten Relation Endpoints wird in der Visualisierung farbig hervorgehoben. Den Fokus kann der Benutzer auf mehrere Art und Weise ändern.

Beispielsweise kann er ihn über die Combobox mit der Beschriftung *Focused Relation Endpoint* wechseln. In ihr stehen alle Chapters und References des Documents zur Auswahl. Da ein Document tendenziell viele Chapters und References besitzt und die Auswahlliste der Combobox entsprechend lang werden kann, ist es möglich, ihre Einträge anhand ihrer Bezeichner zu filtern. Möchte der Benutzer beispielsweise ein Chapter fokussieren, welches in seinem vollständigen Namen die Zeichenkette „Design“ verwendet, trägt er diese in die Combobox ein. Bei jeder Tastatureingabe wird überprüft, welche Relation Endpoints das Kriterium erfüllen. Lediglich diese Relation Endpoints werden anschließend in der Auswahlliste der Combobox angezeigt. Wählt der Benutzer einen Eintrag aus der Liste aus, wird der damit verbundene Relation Endpoint fokussiert und der Graph entsprechend aktualisiert. Eine weitere Möglichkeit den Fokus auf einen Relation Endpoint zu setzen ist die, einen Doppelklick mit der linken Maustaste auf dem entsprechenden Knoten auszuführen. Letztendlich kann der Benutzer einen Relation Endpoint auch über einen Eintrag in dessen Kontextmenü fokussieren.

Das Kontextmenü einer Kante bzw. eines Knotens kann über einen Rechtsklick auf den Knoten bzw. die Kante geöffnet werden. Im Fall einer Kante enthält es nahezu dieselben Einträge wie das Kontextmenü des *Relations*-Reiters (siehe Abbildung 5.2(b)). Da sich die Interaktionsmöglichkeiten von Chapters und References grundlegend unterscheiden, besitzen die entsprechenden Knotentypen im Graph jeweils ein spezialisiertes Kontextmenü.

Über die Schaltflächen rechts neben dem Graphen kann der Benutzer in ihn hinein- oder aus ihm herauszoomen, seine Größe an die Zeichenfläche anpassen oder die angezeigten Knoten und Kanten automatisiert anordnen. Das Verhalten der Visualisierung kann über

die ⚙-Schaltfläche angepasst werden. Die Zeichenfläche kann über Drag-and-Drop beliebig verschoben werden.

Ebenso wie die Einträge der Tabellen besitzen die Knoten und Kanten des Graphen Tooltips, welche die Detailinformationen von Chapters bzw. von References oder von Relations anzeigen.

5.4 Anlegen einer Relation

In diesem Kapitel wird exemplarisch gezeigt, wie eine Relation in UniMoDoc angelegt werden kann. Im Beispiel möchte der Benutzer eine Relation zwischen dem *Test*-Chapter und einer noch anzulegenden Reference erstellen. Die Reference soll eine vorhandene JUnit-Testklasse repräsentieren und die Relation den *refers-to* Relation Type verwenden. Von diesem wird angenommen, dass er im Document noch nicht existiert.

Abbildung 5.4 zeigt den *Add Relation*-Dialog, welchen der Benutzer zum Anlegen der Relation geöffnet hat. Der Dialog lässt sich von mehreren Stellen des Programms aus öffnen. Beispielsweise aus dem *Relations*-Reiter, aus der Visualisierung oder aus bestimmten Widgets. Gleiches gilt auch für die weiter unten vorgestellten Dialoge zum Anlegen von Relation Types (*Add Relation Type*-Dialog) und von References (*Add Reference*-Dialog). Da es sich bei dem Relation Type bzw. dem Ziel der Relation um eine Pflichtangabe handelt, wird der Benutzer im Dialog darauf hingewiesen, dass er einen Relation Type bzw. ein Ziel auswählen muss. Die Relation kann erst dann hinzugefügt werden, wenn alle verpflichtenden Eigenschaften angegeben wurden. Einen neuen Relation Type bzw. eine neue Reference kann der Benutzer über die Schaltfläche neben der Relation Type- bzw. neben der Target-Combobox erstellen. Im Beispiel verwendet der Benutzer diese Schaltflächen, um zunächst den *refers-to* Relation Type und anschließend die Reference anzulegen.

Beim Klicken auf die -Schaltfläche öffnet sich der in Abbildung 5.5(a) abgebildete *Add Relation Type*-Dialog. In diesem kann der Benutzer neben dem Namen, der Kategorie und der Beschreibung des anzulegenden Relation Types dessen Darstellungsinformationen angeben. Letztere werden in einer Vorschau visualisiert. Nachdem der Benutzer die in der Abbildung zu sehenden Werte eingetragen hat, legt er den *refers-to*-Relation Type über einen Klick auf die *Add*-Schaltfläche an. Der angelegte Relation Type wird anschließend in der Relation Type-Combobox des wieder sichtbaren *Add Relation*-Dialogs (siehe Abbildung 5.4) ausgewählt.

Da im Beispiel noch die Reference angelegt und als Ziel der Relation ausgewählt werden muss, klickt der Benutzer auf die -Schaltfläche des *Add Relation*-Dialogs. Beim Klick auf diese öffnet sich der *Add Reference*-Dialog aus Abbildung 5.5(b). Die Eigenschaften der anzulegenden Reference sind im Dialog in drei Reitern organisiert. Im Reiter *General* sind die wichtigsten Eigenschaften der Reference, wie beispielsweise der Name oder der Standort enthalten. Die übrigen Eigenschaften sind auf die Reiter *Details* und *Comments* verteilt. Im Beispiel wählt der Benutzer die Testklasse *RelationListBeanTest.java* über den Dateiauswahldialog der *Browse*-Schaltfläche aus. Hat der Benutzer die Datei ausgewählt, werden einige Metadaten der Klasse (z. B. der Name, die Beschreibung, der Autor, die

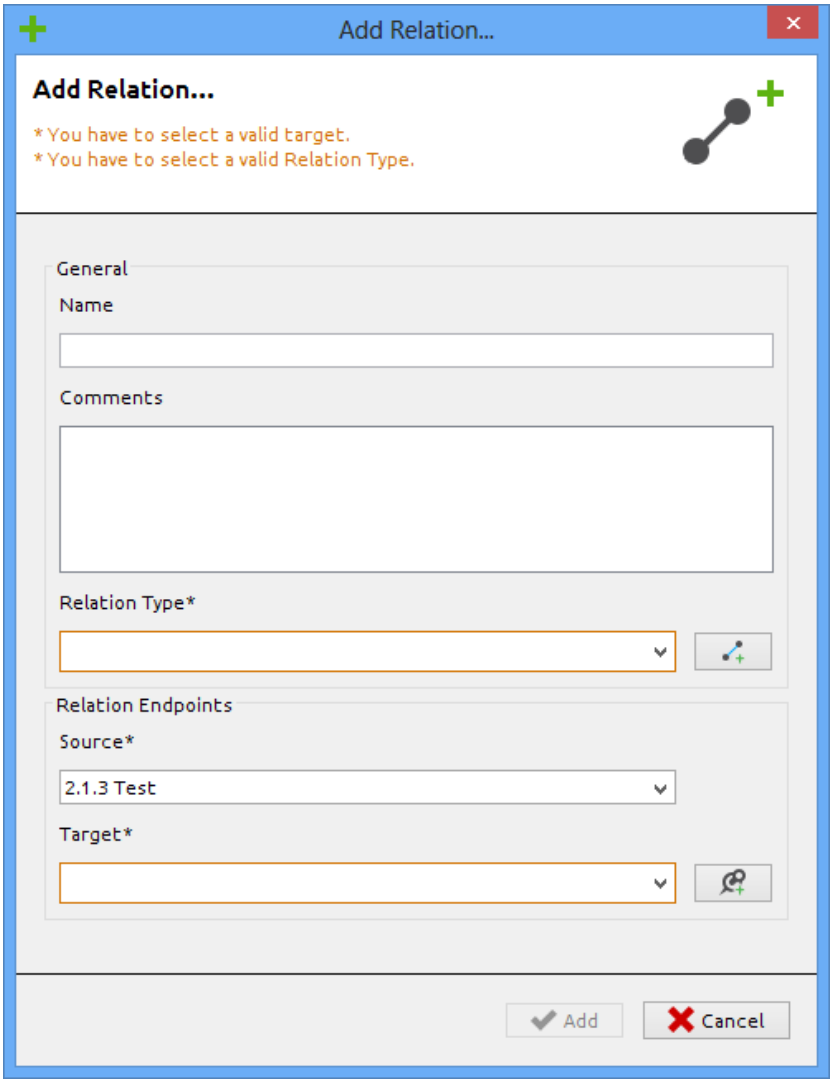
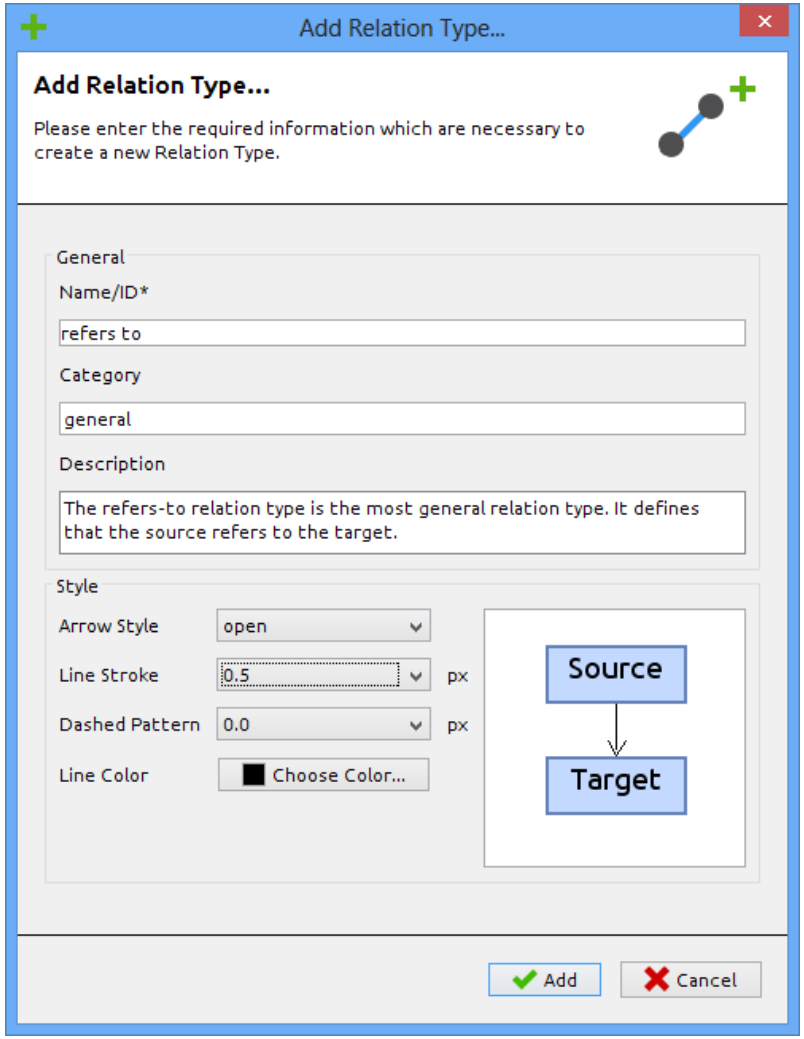


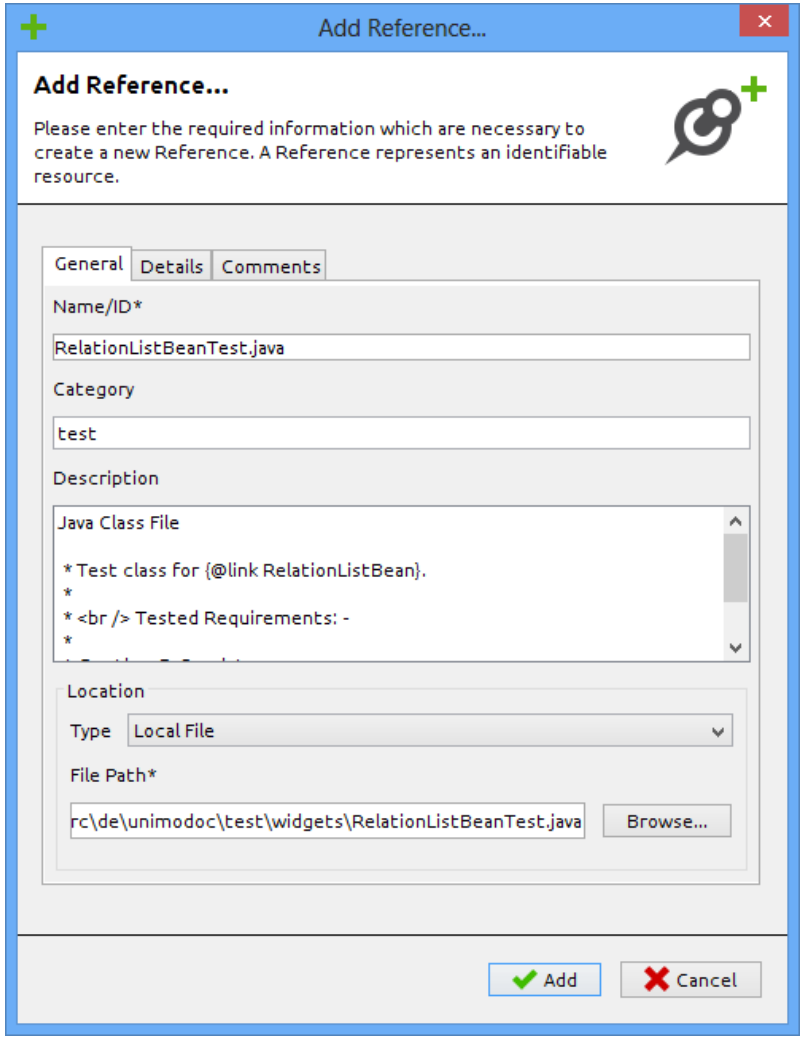
Abbildung 5.4: *Add Relation*-Dialog

Version,...) aus dem Javadoc-Kommentar der Klasse und der Klasse selbst ausgelesen und in die entsprechenden Felder des Dialogs eingetragen. Somit muss der Benutzer in diesem Beispiel nur die Kategorie der Reference manuell eintragen. Die definierte Reference legt er anschließend über einen Klick auf die *Add*-Schaltfläche des Dialogs an. Nachdem sich der *Add Reference*-Dialog geschlossen hat, erscheint wiederum der *Add Relation*-Dialog, in welchem nun die angelegte Reference als Ziel ausgewählt ist.

Da nun alle Pflichtfelder des *Add Relation*-Dialogs ausgefüllt sind, kann der Benutzer die darin definierte Relation anlegen.



(a) *Add Relation Type*-Dialog



(b) *Add Reference*-Dialog

Abbildung 5.5: *Add Relation Type*- und *Add Reference*-Dialog

6 Evaluation

Dieses Kapitel beschreibt, wie das in dieser und der Arbeit von [Pankratz, 2013] entstandene Werkzeug, UniMoDoc, und die darin umgesetzten Konzepte durch die Autoren evaluiert wurde. Zunächst wird in Kapitel 6.1 die Zielsetzung der Evaluation definiert. In Kapitel 6.2 wird anschließend darauf eingegangen, welche Methode zur Evaluation ausgewählt wurde. Darin ist auch beschrieben, welche alternativen Methoden zur Auswahl standen und welche Nachteile diese gegenüber der gewählten Methode, nämlich der über ein kontrolliertes Experiment, aufweisen. Ausschlaggebend für den Entwurf eines Experiments sind dessen Rahmenbedingungen. Diese werden in Kapitel 6.3 genannt. Anschließend werden in Kapitel 6.4 die konkreten Fragen definiert, welche aus der Zielsetzung abgeleitet wurden. In Kapitel 6.5 wird der Experimententwurf im Detail erläutert. Die Bedrohungen der Gültigkeit des Experimententwurfs werden im Kapitel 6.6 behandelt. Kapitel 6.7 beschreibt darauf den Test und die Durchführung des Experiments. Abschließend werden in Kapitel 6.8 die Experimentergebnisse erläutert und in Kapitel 6.9 Schlussfolgerungen aus diesen gezogen. Der Aufbau und die Inhalte des Kapitels wurde von der in [Prechelt, 2001] vorgeschlagenen Struktur zur Dokumentation von kontrollierten Experimenten inspiriert.

6.1 Zielsetzung

Das Ziel der Evaluation war es, zu prüfen, ob das entstandene Werkzeug, UniMoDoc, und die darin umgesetzten Konzepte einen Mehrwert bei der Dokumentation von Modulen gegenüber konventionellen Dokumentationswerkzeugen (z. B. Textverarbeitungswerkzeuge, Code-Dokumentationsgeneratoren) bietet.

Der Mehrwert sollte in der Evaluation objektiv gemessen werden.

6.2 Evaluationsmethode

Die Zielsetzung aus dem vorherigen Kapitel wurde über ein kontrolliertes Experiment evaluiert. Unter einem kontrollierten Experiment wird nach [Prechelt, 2001] eine Studie verstanden, in der alle für das Experimentergebnis relevanten Umstände konstant gehalten werden. Diese Umstände werden als *Störvariablen* bezeichnet, da sie sich – falls nicht angemessen kontrolliert – ungewollt auf die Ergebnisse des Experiments auswirken können. Nicht zu den Störvariablen werden die Umstände gezählt, die im Experiment untersucht

werden sollen. Sie werden als *unabhängige Variablen* bezeichnet. Durch die kontrollierte Änderung der unabhängigen Variablen werden unterschiedliche Experimentergebnisse erzielt. Diese können miteinander verglichen und dadurch Schlüsse hinsichtlich der Auswirkung der unabhängigen Variablen gezogen werden. Da die Ergebnisse von den unabhängigen Variablen direkt abhängen, werden sie auch als *abhängige Variablen* bezeichnet.

Sofern ein kontrolliertes Experiment angemessen geplant, ausgeführt und dokumentiert wird, lässt es zum einen sehr genaue und zuverlässige Schlussfolgerungen über die Auswirkung der unabhängigen Variablen zu. Und zum anderen lässt es sich dadurch leicht nachvollziehen und reproduzieren.

Neben einem kontrollierten Experiment sind weitere empirische Forschungsmethoden für die Evaluation denkbar gewesen. Diese werden im Folgenden aufgezählt und deren signifikantesten Nachteile gegenüber einem kontrollierten Experiment genannt. Die genannten Nachteile waren ausschlaggebend bei der Wahl der Evaluationsmethode.

Fallstudie Anders als bei kontrollierten Experimenten ist die Umgebung in einer Fallstudie nicht exakt vorgeschrieben. Daher können die Ergebnisse einer mehrfach durchgeführten Fallstudie in Abhängigkeit der Umgebung, in der sie durchgeführt wurde, variieren. In diesen Fällen ist es schwierig die Ergebnisse der Durchläufe gemeinsam zu analysieren und dadurch zufällige oder einmalige Effekte zu bestimmen.

Feldstudie Feldstudien sind Studien, die im Feld, d. h. unter realen und daher meist komplexen Bedingungen durchgeführt werden. Diese Bedingungen sind – selbst wenn sie exakt dokumentiert sind – für andere Forscher praktisch nicht reproduzierbar. Da die Bedingungen eines kontrollierten Experiments durch das Experiment selbst vorgegeben werden, kann dieses – falls entsprechend entworfen und dokumentiert – vergleichsweise einfach reproduziert werden.

Umfrage Über Umfragen kann lediglich der subjektive Eindruck der Befragten erfasst werden. Entsprechend schwierig gestaltet sich eine Interpretation der Antworten. In kontrollierten Experimenten hingegen können den Probanden Aufgaben gestellt und deren Antworten anschließend objektiv ausgewertet werden.

6.3 Rahmenbedingungen

Ein wichtiger Aspekt bei dem Entwurf eines Experiments sind die Rahmenbedingungen, unter welchen das Experiment durchgeführt wird. Diese haben unter anderem Einfluss darauf, welche konkreten Fragen im Experiment realistischerweise beantwortet werden können. Das Experiment wurde unter Berücksichtigung folgender Rahmenbedingungen (RB) entworfen:

RB-1 Das Experiment sollte im Rahmen einer 90-minütigen Übung der Vorlesung „Qualitätssicherung und Wartung (QSW)“ der Universität Stuttgart mit den teilnehmenden Masterstudenten – den Probanden – durchgeführt werden.

Dadurch ergab sich eine zeitliche Obergrenze von 90 Minuten für das Experiment.

RB-2 Die Probanden waren Masterstudenten des Studiengangs Softwaretechnik. Daher konnte angenommen werden, dass sie über Grundkenntnisse in der Programmiersprache Java verfügen.

RB-3 Es nahmen 12 Masterstudenten an der QSW-Vorlesung regelmäßig teil. Daher konnte mit maximal dieser Anzahl an Probanden gerechnet werden.

RB-4 Das Experiment sollte in einem Rechnerpool der Universität Stuttgart durchgeführt werden. Entsprechend wurde bei der Experimentplanung angenommen, dass den Probanden Rechner zur Verfügung stehen.

6.4 Experimentfragen

Aus der Zielsetzung (siehe Kapitel 6.1) lassen sich folgende Experimentfragen (EF) ableiten, welche im Experiment geklärt werden sollten:

EF-1 Können die Probanden mithilfe einer in UniMoDoc erstellten Moduldokumentation Fragen zu dem Softwareprojekt, das sie dokumentiert, effizienter und effektiver beantworten, als lediglich mithilfe der Entwicklungsartefakte (ausschließlich der Moduldokumentation) des Projekts?

EF-2 Tragen die explizit definierbaren Beziehungen und deren Visualisierung zum Verständnis der Beziehungen, die in einer Moduldokumentation existieren, bei?

Die Umsetzbarkeit der in Kapitel 3.4 vorgestellten Konzepte wurde durch deren technische Umsetzung in Form von UniMoDoc belegt. In der Evaluation soll nun die tatsächliche Tauglichkeit der Konzepte anhand des entstandenen Programms evaluiert werden. Im Experiment wird hierzu unter anderem der subjektive Eindruck der Probanden in Bezug auf UniMoDoc erfasst. Dieser könnte durch bestimmte Faktoren, welche keine Aussagekraft über die Konzepte haben sondern lediglich mit deren Umsetzung zusammenhängen, negativ beeinflusst werden. Zu diesen Faktoren zählen vor allem die Stabilität und die Benutzerfreundlichkeit von UniMoDoc. Um solche potenziellen Effekte feststellen zu können, werden zusätzlich folgende Experimentfragen definiert:

EF-3 Wird UniMoDoc während dem Experiment abstürzen?

EF-4 Empfinden die Probanden UniMoDoc bei der Bearbeitung der Aufgaben als benutzerfreundlich?

Eine weitere mögliche Experimentfrage wäre außerdem gewesen, ob das in dieser Arbeit erstellte Template zur Verwaltung von Testinformation geeignet ist. Analog hätte auch eine Experimentfrage hinsichtlich des von [Pankratz, 2013] erstellten Templates formuliert werden können. Da mangels Fachwissen der Probanden jedoch nicht damit zu rechnen war,

dass diese Fragestellungen von ihnen repräsentativ beantwortet werden konnten, wurden die Fragestellungen nicht in die Liste der Experimentfragen aufgenommen.

Die initiale Planung des Experiments sah eine weitere Experimentfrage vor. Das Ziel dieser Frage war es zu ermitteln, ob mithilfe der Templates, welche in dieser und der Arbeit von [Pankratz, 2013] entstanden sind, detailliertere und besser strukturierte Moduldokumentationen von den Probanden erstellt werden, als ohne die Templates. Die Experimentfrage wurde jedoch verworfen, da die Rahmenbedingungen keine angemessene Beantwortung der Frage zuließ. Um die Frage angemessen zu beantworten, wäre es notwendig gewesen, die Probanden Module dokumentieren zu lassen, die sie davor selbst erstellt haben. Angesichts des zeitlichen Rahmens von 90 Minuten ist das im Experiment jedoch nicht möglich.

Die Experimentfragen bilden die Grundlage für die konkreten Fragen und Aufgaben, welche den Probanden während des Experiments gestellt werden.

6.5 Experimententwurf

In diesem Kapitel wird der Entwurf des Experiments im Detail erläutert. In Kapitel 6.5.1 wird eine Übersicht über die Abschnitte des Experiments gegeben, welche in den darauffolgenden Kapiteln (siehe Kapitel 6.5.2 bis Kapitel 6.5.6) im Detail erläutert werden.

6.5.1 Kurzübersicht

Das Experiment gliedert sich in die Phasen *Einleitung*, *erster Experimentabschnitt*, *erste Feedbackrunde*, *zweiter Experimentabschnitt* und *zweite Feedbackrunde*. Diese werden im Folgenden kurz erläutert. Die Aufzählungsreihenfolge entspricht der chronologischen Ausführungsreihenfolge der Phasen. Die Minutenangabe, welche dem jeweiligen Bezeichnern der Phase folgt, gibt die Dauer der Phase an.

Einleitung (5 min) In der Einleitungsphase wird den Probanden das Ziel, der Kontext und der zeitliche Ablauf des Experiments erläutert. Des Weiteren werden organisatorische Hinweise gegeben.

Erster Experimentabschnitt (40 min) Im ersten Experimentabschnitt bearbeiten die Probanden Aufgaben, in welchen Fragen hinsichtlich eines realen Softwareprojekts gestellt werden. Beim Bearbeiten der einen Hälfte der Aufgaben verwenden die Probanden ausschließlich die Moduldokumentation des Projekts, welche in UniMoDoc erstellt wurde. Die zweite Hälfte der Aufgaben wird nur mithilfe der Entwicklungsartefakte ausschließlich der Moduldokumentation bearbeitet. Die Probanden werden in diesem Abschnitt in vier Gruppen aufgeteilt. Bei dem Softwareprojekt handelt es sich um UniMoDoc selbst. Die Lösungen der Probanden werden dazu verwendet, um objektive Schlüsse hinsichtlich der Experimentfragen EF-1 und EF-2 zu ziehen.

Erste Feedbackrunde (10 min) In der ersten Feedbackrunde wird mithilfe eines Feedbackbogens ermittelt, welchen Einfluss der Einsatz von UniMoDoc auf die Bearbeitungsgeschwindigkeit der Probanden beim Lösen der einzelnen Aufgaben hatte. Des Weiteren werden Fragen zur Stabilität und zur Benutzerfreundlichkeit gestellt, welche die Probanden beim Bearbeiten der Aufgaben erfahren haben. Im Feedbackbogen werden solche Fragen gestellt, welche zur Beantwortung der Experimentfragen EF-1 bis EF-4 beitragen.

Zweiter Experimentabschnitt (15 min) Im zweiten Experimentabschnitt erweitern die Probanden eine in UniMoDoc erstellte Moduldokumentation mithilfe von UniMoDoc. Dabei sollen sie drei neue Sections mit unterschiedlichen Widgets anlegen und mit Inhalt befüllen. Außerdem sollen sie einen Relation Type, zwei References und zwei Relations anlegen. Dieser Abschnitt dient als Vorbereitung für die zweite Feedbackrunde.

Zweite Feedbackrunde (10 min) In der zweiten Feedbackrunde werden die Probanden um Feedback hinsichtlich der Stabilität und der Benutzerfreundlichkeit von UniMoDoc geben, welche sie beim Bearbeiten der Aufgaben des zweiten Experimentabschnitts wahrgenommen haben. Außerdem werden sie gefragt, welches Werkzeug sie vorziehen würden, um Module zu dokumentieren. Analog dazu werden sie gefragt, welche Form der Moduldokumentation sie vorfinden wollen würden, sofern sie eine fremde Software weiterentwickeln müssten. Abschließend werden ihnen Fragen zu ihren Vorkenntnisse gestellt. Die Antworten werden hauptsächlich dazu verwendet, um die Experimentfragen EF-3 und EF-4 zu beantworten.

6.5.2 Einleitung

Zu Beginn der Einleitungsphase erhält jeder Proband eine Mappe, welche ein Einleitungsdokument und ein Faltblatt enthält.

Das Einleitungsdokument beschreibt das allgemeine Ziel und den Kontext des Experiments. Des Weiteren informiert es die Probanden über dessen Dauer und dessen groben zeitlichen Ablauf. Außerdem werden im Dokument wichtige organisatorische Hinweise gegeben. Beispielsweise werden die Probanden dazu aufgefordert über die volle Zeit des Experiments an diesem teilzunehmen, da ihre Ergebnisse ansonsten unbrauchbar werden. Neben den Hinweisen enthält das Informationsblatt einen Link zu einem Archiv, das über das Internet von den Probanden heruntergeladen und anschließend entpackt wird. Das Archiv enthält eine ausführbare UniMoDoc.jar-Datei und die Materialien, welche für die weiteren Phasen des Experiments relevant sind.

Das Faltblatt eines Probanden wird an dessen Arbeitsplatz aufgestellt und identifiziert ihn mithilfe einer Nummer zwischen eins und zwölf. Jede Nummer ist nur auf genau einem der Faltblätter abgebildet. Sie wird im weiteren Verlauf als *Faltblattnummer* bezeichnet. Die Nummer wird in die Aufgabenblätter und die Feedbackbögen eingetragen, so dass diese einem Probanden eindeutig zugeordnet werden können. Durch die Faltblattnummer ist es den Experimentatoren außerdem möglich, während des Experiments zu bestimmen zu welcher Gruppe ein Proband gehört. Dadurch können die Aufgabenblätter, die für

die unterschiedlichen Gruppen in Inhalt und Reihenfolge variieren, zuverlässig ausgeteilt werden. Aus der Nummer lässt sich die Gruppenzugehörigkeit eines Probanden wie folgt ableiten: <*Nummer*> modulo 4 = <*Nummer der Gruppe*>.

Die Mappen werden zufällig und mit fortlaufender und aufsteigender Faltblattnummer an die Probanden ausgegeben. Durch letzteres wird sichergestellt, dass die Probandengruppen ähnlich groß sind. Jeder Proband hat fünf Minuten Zeit, um sich das Einleitungsdokument durchzulesen und das Faltblatt an seinem Arbeitsplatz sichtbar aufzustellen.

6.5.3 Erster Experimentabschnitt

In diesem Kapitel wird der erste Experimentabschnitt im Detail erläutert. Da dieser sehr komplex und umfangreich ist, wird er in mehrere, inhaltlich spezifische Kapitel unterteilt. Als erstes wird der Aufbau des Abschnitts beschrieben. Der letztendliche Entwurf des Experimentabschnitt wird dabei schrittweise hergeleitet und die jeweiligen Schritte begründet. Anschließend wird auf die Fragen- und Aufgabentypen des Experimentabschnitts eingegangen. Darauf folgt die Beschreibung der Materialien und Werkzeuge, welche den Probanden zur Verfügung gestellt werden. Abschließend wird der zeitliche Ablauf des Abschnitts erläutert.

Aufbau

Das Ziel des ersten Experimentabschnitts ist es festzustellen, ob der Einsatz von UniMoDoc und einer darin erstellten Moduldokumentation die Beantwortung von Fragen in Bezug auf das dokumentierte Projekt erleichtert. Um diese Frage adäquat beantworten zu können, wird ein Vergleich benötigt. Dieser kann mithilfe einer Kontrollgruppe erreicht werden, welche dieselben Fragen lediglich mithilfe der Entwicklungsartefakte des Projekts ausgenommen der Moduldokumentation beantwortet. Durch die Kontrollgruppe ist es den Experimentatoren möglich, die Antworten der beiden Gruppen objektiv zu vergleichen. Dieser vorläufige Experimentabschnittsentwurf kann in der Notation von [Prechelt, 2001] wie folgt notiert werden:

$G_0 : F / UniMoDoc$
$G_1 : F / !UniMoDoc$

G_0 bezeichnet die Gruppe der Probanden, welche UniMoDoc und die darin erstellte Moduldokumentation verwendet um die Fragen F zu beantworten. Analog dazu bezeichnet G_1 die Gruppe der Probanden, welche nicht UniMoDoc ($!UniMoDoc$), sondern lediglich die Entwicklungsartefakte des Projekts verwendet um die Fragen F zu beantworten.

Neben dem objektiven Vergleich durch die Experimentatoren ist ein subjektiver Vergleich durch die Probanden wünschenswert. Entsprechend benötigen die Probanden selbst eine Vergleichsgrundlage. Diese wird ihnen geschaffen, indem sie die Fragen F einmal mithilfe von UniMoDoc und der darin erstellten Moduldokumentation (Fall 1) beantworten und einmal

ohne (Fall 2). Diese Entscheidung birgt jedoch auch Gefahren an die innere Gültigkeit des Experiments. Das Experiment kann aus zwei Gründen an Glaubwürdigkeit verlieren. Zum einen ist mit einem Lerneffekt seitens der Probanden von Fall 1 zu Fall 2 zu rechnen. Dieser kann gemindert werden, indem die Probanden nicht dieselben Fragen sondern Fragen F' vom selben Fragentyp gestellt bekommen. Beispielsweise sind zwei Fragen vom selben Fragentyp, wenn sie nach den Abhängigkeiten eines Moduls M fragen, wobei sich M für jede der Fragen unterscheidet. Zum anderen ist nicht auszuschließen, dass die Reihenfolge der Bearbeitung einen Einfluss auf die Antworten der Probanden hat. Beide Probleme können dadurch entschärft werden, dass für alle möglichen Kombinationen der Bearbeitungsreihenfolge, der Fragen und der verwendbaren Werkzeuge jeweils eine Gruppe vorgesehen wird:

$G_0 : F/UniMoDoc$	$F'/!UniMoDoc$
$G_1 : F/!UniMoDoc$	$F'/UniMoDoc$
$G_2 : F'/UniMoDoc$	$F/!UniMoDoc$
$G_3 : F'/!UniMoDoc$	$F/UniMoDoc$

Die Bedeutung der Zeilen wird nachfolgend anhand der ersten Zeile erläutert. Die Probanden der Gruppe G_0 verwenden UniMoDoc und die darin erstellte Moduldokumentation, um die Fragen F in Bezug auf das dokumentierte Softwareprojekts zu beantworten. Anschließend verwenden sie ausschließlich die Entwicklungsartefakte des Softwareprojekts ($!UniMoDoc$), um die Fragen F' zu beantworten.

Fragentypen

Folgende Fragentypen bzw. Aufgabentypen werden gestellt:

F-1 Durch welches Modul bzw. welche Module wird die Anforderung A realisiert? Geben Sie den Modulnamen je Modul an.

F-2 Geben Sie alle JUnit-Testklassen an, welche die Anforderung A testen. Hierbei genügt der einfache Name der Klasse (z. B. MyTest.java). Geben Sie außerdem die Anzahl der Methoden je Klasse an.

F-3 Es gibt genau ein Modul, das die Anforderung A realisiert. Vervollständigen Sie die Beschreibung zu dieser Anforderung, indem Sie die komplette Beschreibung finden: <*Ein Teil der Beschreibung von A*>...

F-4 Die Klasse C, die zum Modul M gehört, wurde in mehreren Testläufen ausgeführt und die Ergebnisse je Testlauf durch HTML-Dateien protokolliert. Geben Sie das Datum und die Uhrzeit der Testläufe zur Testklasse an. Geben Sie außerdem die Anzahl der Fehler (Failures) je Testlauf an.

F-5 Welche Abhängigkeiten zu anderen Modulen hat das Modul M? Geben Sie die Namen der Module an, welche von M abhängen (Ingoing Dependencies). Geben Sie außerdem die Namen der Module an, von welchen M abhängt (Outgoing Dependencies).

Materialien

Wie oben bereits erwähnt, handelt es sich bei dem Softwareprojekt, zu welchem Fragen gestellt werden, um UniMoDoc. Das heißt, dass die Probanden entweder die Moduldokumentation oder die Entwicklungsartefakte von UniMoDoc zur Beantwortung der Fragen benutzen. Sowohl die Moduldokumentation, als auch die Entwicklungsartefakte sind im digitalen Archiv enthalten, welches die Probanden in der Einführungsphase heruntergeladen und entpackt haben. Im Folgenden wird zunächst auf die Entwicklungsartefakte und anschließend auf die Moduldokumentation eingegangen.

Zu den Entwicklungsartefakten zählt der Quellcode, die Javadoc-Dokumentation, die Testfälle und die Testprotokolle von UniMoDoc. Des Weiteren umfassen sie ein PDF-Dokument, in dem die Anforderungen, welche an UniMoDoc gestellt werden, aufgelistet und beschrieben sind. Bei den Testfällen handelt es sich um JUnit-Testklassen und bei den Testprotokollen um HTML-Dateien. Die Testklassen wurden in einem Ant-Buildskript automatisiert ausgeführt und die Ergebnisse automatisiert protokolliert. Dazu wurden die Ant-Tasks `junit` und `junitreport` eingesetzt. Die Testklassen sind in Java-Packages organisiert. Packages, die nur Testklassen enthalten, werden als Test-Packages bezeichnet. Jedes dieser Test-Packages kann genau einem nicht-Test-Package eindeutig zugeordnet werden. Am Beispiel eines Packages mit dem Namen „Model“ bedeutet das, dass genau ein Test-Package existiert, welches nur Testklassen enthält, die Klassen aus dem „Model“-Package testen.

Die Module von UniMoDoc wurden so gewählt, dass sie den Java-Packages des Quellcodes entsprechen. Dadurch können die Module im Quellcode über `package-info.java`-Dateien dokumentiert werden. Diese Art der Modularisierung hat zur Folge, dass in der Moduldokumentation für jedes Package (auch die Test-Packages) ein Modul dokumentiert wird. Die Informationen eines Moduls werden in ihr über eine angepasste Form des Template von [Pankratz, 2013] erfasst. Zu jedem Modul werden unter anderem folgende Informationen festgehalten:

- die Anforderungen, welche das Modul realisiert.
- die Abhängigkeiten zu anderen Modulen.

Die Abhängigkeiten werden mithilfe von Relations und den Relation Types *is-part-of*, *depends-on* und *tested-by* dargestellt. Um auch die Anforderungen über Relations mit den Modulen verknüpfen zu können, wurden diese als eigenständige Chapters in die Dokumentation aufgenommen. Die Relations zwischen den Anforderungen und den Modulen verwenden den Relation Type *realized-by*.

Module, welche Test-Packages repräsentieren, werden über eine angepasste Form des in Kapitel 3.4.5 vorgestellten Templates zur Verwaltung von Testinformationen dokumentiert. In jedem dieser Module werden unter anderem die Testmittel (engl. testware) aufgelistet. Zu diesen zählen die Testklassen und die Testprotokolle. Zwischen den Testklassen und den Testprotokollen der jeweiligen Testklasse sind Relations vom Typ *protocolled-by* definiert. Des Weiteren sind die Testklassen über Relations des Typs *tested-by* mit den Anforderungen verbunden, die sie testen.

Alle Informationen, welche sich in der Moduldokumentation finden lassen, sind auch in den Entwicklungsartefakten enthalten. In Abhängigkeit der Information lässt sich die Information im Anforderungsdokument, im Quellcode bzw. der Javadoc-Dokumentation, den Testfällen oder den Testprotokollen finden.

Werkzeuge

Im Folgenden werden die konkreten Werkzeuge mitsamt ihrem Verwendungszweck aufgelistet, die von den Probanden verwendet werden dürfen:

UniMoDoc: UniMoDoc als ausführbare JAR-Datei zum Verwenden der Moduldokumentation; den Standardbrowser des Rechners zum Anzeigen der Testprotokolle;

!*UniMoDoc*: das Dateisystem zur Navigation; den Standardbrowser des Rechners zum Anzeigen der Testprotokolle und der Javadoc-Dokumentation; den Standardtexteditor des Rechners zum Anzeigen der Quellcodedateien;

Organisation und Ablauf

Für alle Fragen-Werkzeug-Paare ($F/UniMoDoc$, $F'/UniMoDoc$, $F/!UniMoDoc$ und $F'/!UniMoDoc$) existiert jeweils ein Aufgabenblatt. Die Blätter listen die jeweiligen Fragen auf und bieten Platz für deren Antworten. Des Weiteren klären sie die Probanden darüber auf, wie viel Zeit sie für die Bearbeitung der Aufgaben haben und welche konkreten Werkzeuge sie verwenden dürfen. Zusätzlich werden Hinweise zur Verwendung von UniMoDoc bzw. zu den Entwicklungsartefakten gegeben.

Am Anfang des ersten Experimentabschnitts erhält jeder Proband ein Aufgabenblatt (in Abhängigkeit seiner Faltblattnummer) in ausgedruckter Form. Die Bearbeitungszeit des Aufgabenblatts beträgt 20 Minuten. Nach Ablauf der Bearbeitungszeit werden die Probanden dazu aufgefordert das Aufgabenblatt in die Mappe zu legen, welche sie in der Einleitungsphase erhalten haben. Anschließend erhalten sie das zweite Aufgabenblatt (in Abhängigkeit ihrer Faltblattnummer) in ausgedruckter Form. Dessen Bearbeitungsdauer beträgt wiederum 20 Minuten. Genauso wie das erste Aufgabenblatt wird das zweite nach dem Ablauf der Bearbeitungszeit in die Mappe gelegt.

6.5.4 Erste Feedbackrunde

Zu Beginn der ersten Feedbackrunde erhält jeder Proband den ersten Feedbackbogen. Darin werden die Probanden gefragt, ...

... ob sie die Aufgaben mit oder ohne UniMoDoc besser lösen konnten.
(Auswahl: 1=*sehr viel besser ohne* bis 7=*sehr viel besser mit*)

... ob sie die Visualisierung von UniMoDoc zur Bearbeitung der Aufgaben vom Typ F-1 bis F-5 mindestens einmal eingesetzt haben. (Auswahl: *Ja* | *Nein*)

... und falls ja, welche Auswirkung der Einsatz auf ihre Bearbeitungsgeschwindigkeit hinsichtlich der jeweiligen Aufgabe hatte. (Auswahl: 1=*viel langsamer* bis 5=*viel schneller* und zusätzlicher Option *Ich habe die Visualisierung dafür nicht verwendet*)

... welche Funktionen besonders hilfreich waren. (Freitext)

... welche Funktionen sie vermisst haben. (Freitext)

... welche Funktionen verbesserungsfähig sind. (Freitext)

... wie oft UniMoDoc abgestürzt ist. (Auswahl: *0* | *1-5* | *5-10* | *>10*)

... wie sie die Benutzerfreundlichkeit von UniMoDoc werten.
(Auswahl: 1=*nicht benutzerfreundlich* bis 7=*sehr benutzerfreundlich*)

... ob sie sonstige Kommentare abgeben möchten. (Freitext)

Die Inhalte der Klammern geben die Antwortart der jeweiligen Frage und – sofern vorhanden – die zugehörigen Antwortmöglichkeiten an. Bei allen Auswahlfragen soll nur eine Antwort von den Probanden ausgewählt werden. Die erste Feedbackrunde ist auf zehn Minuten beschränkt.

6.5.5 Zweiter Experimentabschnitt

In diesem Experimentabschnitt soll eine bereits vorhandenen Moduldokumentation erweitert werden. Das Ziel davon ist die Benutzerfreundlichkeit und die Stabilität von UniMoDoc in diesem Anwendungsfall zu messen. Da hierzu weder ein objektiver Vergleich seitens der Experimentatoren, noch ein subjektiver Vergleich seitens der Probanden nötig ist, wird keine Einteilung in Gruppen vorgenommen. Entsprechend bearbeiten alle Probanden dieselbe Aufgabe zur selben Zeit.

Am Anfang des zweiten Experimentabschnitts erhalten die Probanden ein ausgedrucktes Aufgabenblatt, ein ausgedrucktes Informationsblatt und einen USB-Stick. Als Teil der Aufgabenstellung enthält das Aufgabenblatt einen Screenshot vom Soll-Zustand eines Chapters, welcher die zu erweiternde Moduldokumentation zeigt. Alle Informationen, die auf dem Screenshot nicht ersichtlich, aber für die Bearbeitung der Aufgabe nötig sind, werden nach diesem aufgeführt. Um den Soll-Zustand zu erreichen, muss ein Document in UniMoDoc geöffnet und das entsprechende Chapter um jeweils eine Section der Widgets *TextArea*, *RelationList* und *ImageWidget*, um einen Relation Type, um zwei References und um zwei Relations erweitert werden. Das Document und die Dateien, welche durch die References repräsentiert werden sollen, sind im Archiv enthalten, das die Probanden während der Einleitungsphase erhalten haben. Die angelegten Sections sollen abschließend mit den Inhalten befüllt werden, die im Aufgabenblatt implizit oder explizit aufgeführt sind.

Das Informationsblatt enthält Hinweise zur Benutzung von UniMoDoc, welche für die Aufgabe relevant sind. Die Probanden können das Informationsblatt zu rate ziehen, falls sie Schwierigkeiten beim Lösen der Aufgabe haben sollten.

Die Bearbeitungsdauer der Aufgabe beträgt 15 Minuten. Nach Ablauf der Zeit speichern die Probanden das erweiterte UniMoDoc-Document auf dem USB-Stick ab, der ihnen ausgeteilt wurde, und legen diesen in ihre jeweilige Mappe.

6.5.6 Zweite Feedbackrunde

Die zweite Feedbackrunde schließt das Experiment ab. In ihr erhält jeder Proband einen Fragebogen. Die Probanden werden darin gefragt, ...

... welche Funktionen besonders hilfreich waren. (Freitext)

... welche Funktionen sie vermisst haben. (Freitext)

... welche Funktionen verbesserungsfähig sind. (Freitext)

... wie oft UniMoDoc abgestürzt ist. (Auswahl: *0* | *1-5* | *5-10* | *>10*)

... wie sie die Benutzerfreundlichkeit von UniMoDoc werten.
(Auswahl: 1=*nicht benutzerfreundlich* bis 7=*sehr benutzerfreundlich*)

... welches Werkzeug sie verwenden würden, um ein Modul zu dokumentieren.
(Auswahl: *UniMoDoc* | *Textverarbeitugswerkzeug* | *Wiki* und Ergänzungsoption *Sonstiges*)

... welche Form der Moduldokumentation sie vorfinden wollen würden, sofern sie eine fremde Software weiterentwickeln müssten. (Auswahl: *in UniMoDoc* | *in Textform (PDF, Word, usw.)* | *in einem Wiki* und Ergänzungsoption *Sonstiges*)

... welche Erfahrung sie mit Java haben.
(Auswahl: 1=*keine Erfahrung* bis 7=*sehr viel Erfahrung*)

... welche Erfahrung sie mit Javadoc haben.
(Auswahl: 1=*keine Erfahrung* bis 7=*sehr viel Erfahrung*)

... ob sie sonstige Kommentare abgeben möchten. (Freitext)

Die Inhalte der Klammern geben die Antwortart der jeweiligen Frage und – sofern vorhanden – die zugehörigen Antwortmöglichkeiten an. Bei allen Auswahlfragen soll nur eine Antwort von den Probanden ausgewählt werden. Genauso wie die erste Feedbackrunde ist die zweite auf zehn Minuten beschränkt.

6.6 Bedrohungen der Gültigkeit

In den folgenden beiden Kapiteln werden die Bedrohungen an die innere und äußere Gültigkeit erläutert. In Kapitel 6.6.1 werden zu jeder Bedrohung der inneren Gültigkeit die ergriffenen Gegenmaßnahmen beschrieben. In Kapitel 6.6.2 werden die potenziellen Unzulänglichkeiten des Experiments in Bezug auf die externe Gültigkeit erläutert und zu diesen mögliche Gegenmaßnahmen vorgeschlagen.

6.6.1 Innere Gültigkeit

Als *innere Gültigkeit* eines kontrollierten Experiments bezeichnet [Prechelt, 2001] den Grad, in dem die Änderungen in den Werten der abhängigen Variablen nur auf die Änderungen in den unabhängigen Variablen zurückzuführen sind. Der Grad ist umso höher, desto besser die relevanten Störvariablen kontrolliert werden. Eine fehlende Kontrolle der Störvariablen wirkt sich negativ auf die innere Gültigkeit aus. Daher werden die Störvariablen auch als Bedrohung der innere Gültigkeit eines Experiments betrachtet. Im Folgenden werden die möglichen Bedrohungen des geplanten Experiments aufgelistet und in die von [Prechelt, 2001] verwendeten Klassen eingeteilt. Des Weiteren wird zu jeder Bedrohung die Gegenmaßnahme erläutert, welche ergriffen wird. Im Experimententwurf konnten keine Gefahren der Klassen *Instrumentation*, *Historie* und *Regression* festgestellt werden.

Auswahl

In kontrollierten Experimenten werden die Probanden typischerweise in mehrere Gruppen aufgeteilt. Die Gruppenaufteilung kann sich dabei auf die Ergebnisse des Experiments direkt auswirken. Findet die Aufteilung nicht zufällig statt, kann es beispielsweise passieren, dass sich die Gruppen in den Fähigkeiten, der Motivation oder anderen Störvariablen drastisch unterscheiden. Solche Gefahren werden unter dem Begriff *Auswahl* zusammengefasst.
Die Gruppenaufteilung findet in diesem Experiment daher randomisiert statt. Dadurch, dass jeder Proband einer Gruppe per Zufall zugeteilt wird, kann statistisch angenommen werden, dass sich die genannten Einflüsse gleichmäßig auf die Gruppen verteilen.

Reifung

Das Verhalten der Probanden kann sich während der Durchführung eines Experiments ändern. Die Gefahren, welche zu einer Verhaltensänderung führen, werden unter dem Begriff der *Reifung* zusammengefasst. Zu den wichtigsten Reifungseffekten zählen die Ermüdungs-, die Reihenfolge- und die Lerneffekte.
Im ersten Experimentabschnitt können Reihenfolge- und Lerneffekte erkannt werden, indem jeweils eine Gruppe für jede der Permutationen der Reihenfolge, der Fragen (F und F') und der Hilfsmittel ($UniMoDoc$ und $!UniMoDoc$) vorgesehen wird.

Sterblichkeit

Unter der *Sterblichkeit* werden die Gefahren zusammengefasst, welche dazu beitragen, dass ein Proband auf eigenen Wunsch vorzeitig aus dem Experiment ausscheidet.
Die Probanden werden im Vorfeld über das Experiment informiert. Dadurch kann angenommen werden, dass die Probanden, welche zum Experiment erscheinen, auch an diesem über

die volle Dauer teilnehmen wollen. Des Weiteren werden die Probanden auf dem Einleitungsdokument über die Konsequenzen eines vorzeitigen Ausscheidens (für die Experimentatoren) aufgeklärt.

Anforderungscharakteristik

Experimentatoren sind bezüglich der Experimentergebnisse nicht neutral eingestellt, sondern erhoffen sich solche, die ihren Vermutungen entsprechen. Daher kann es beispielsweise passieren, dass die Aufgabenblätter der Gruppen nicht neutral formuliert sind oder die Experimentatoren einzelne Gruppen unbeabsichtigt bevorzugen. Diese Gefahren werden unter der Klasse *Anforderungscharakteristik* zusammengefasst.
Im ersten Experimentabschnitt löst jede Gruppe dieselben Fragentypen einmal mit und einmal ohne UniMoDoc. Somit gibt es keine Gruppe, welche durch die Experimentatoren bevorzugt werden kann. Des Weiteren wird die Einflussnahme der Experimentatoren dadurch auf ein Minimum reduziert, dass die Probanden alle für das Experiment relevanten Informationen in textueller Form enthalten. Um eine Bevorzugung einzelner Gruppen zu vermeiden, enthalten die Aufgabenblätter exakt dieselben Informationen. Ausgenommen davon sind die Informationen, welche unmittelbar die unabhängige Variable (*UniMoDoc* oder !*UniMoDoc*) betreffen.

Verarbeitungsfehler

Während der Durchführung des Experiments oder der Auswertung dessen Ergebnissen können *Verarbeitungsfehler* auftreten. Beispielsweise können die Ergebnisse der Probanden vertauscht werden, verloren gehen oder Daten aus ihnen falsch übertragen werden.
Jeder Proband erhält zu Beginn des Experiments eine Mappe und ein Faltblatt mit einer eindeutigen Nummer. Die Faltblattnummer wird in jedes Aufgabenblatt und jeden Feedbackbogen eingetragen. Außerdem legt jeder Proband seine Ergebnisse in die ihm ausgeteilte Mappe. Durch beide Vorkehrungen ist eine versehentliche Verwechslung oder Vertauschung der Ergebnisse nicht möglich. Fehler in der Auswertung werden dadurch vermieden, dass ein klar definiertes Auswertungsschema festgelegt ist und die Auswertung durch beide Experimentatoren einzeln vorgenommen wird.

6.6.2 Äußere Gültigkeit

Nach [Prechelt, 2001] wird unter der *äußeren Gültigkeit* eines Experiments der Grad verstanden, in welchem sich die Resultate eines Experiments auf andere Anwendungsfälle übertragen lassen. Besonders interessant sind dabei solche, welche in der Praxis vorkommen. Im gesamten Experimententwurf (siehe Kapitel 6.5) wird versucht, die äußere Gültigkeit zu maximieren. Gegeben durch die Rahmenbedingungen des Experiments existieren hierbei jedoch Einschränkungen. Im Folgenden wird auf diese eingegangen.

Im ersten Experimentabschnitt werden Fragen zu einem Softwareprojekt, nämlich UniMoDoc, gestellt. UniMoDoc ist ein reales Softwareprojekt. Entsprechend lassen sich die Ergebnisse des Experiments grundsätzlich auf andere Projekte übertragen. Da UniMoDoc jedoch im Rahmen von zwei Diplomarbeiten entwickelt wurde und somit nicht unbedingt mit ausgereifter und professioneller Software verglichen werden kann, empfiehlt es sich weitere Experimente durchzuführen und dabei andere Softwareprojekte als Grundlage zu nehmen.

Die Fragen des ersten Experimentabschnitts sollen typische Anwendungsfälle simulieren, welche beim Verwenden der Moduldokumentation existieren. Der Fokus der Fragen liegt hierbei auf der Rückverfolgbarkeit (engl. traceability). Da es sich bei den berücksichtigten Anwendungsfällen nur um einen Teil der generell möglichen handelt, kann der Experimententwurf in Zukunft um zusätzliche Anwendungsfälle erweitert werden.

Bei allen Probanden des Experiments handelt es sich um Studenten des Masterstudiengangs Softwaretechnik. Entsprechend können die Ergebnisse des Experiments hinsichtlich anderer Personengruppen nicht ohne Weiteres verallgemeinert werden. Um die externe Gültigkeit zu erhöhen, sollte das Experiment daher mehrfach mit jeweils unterschiedlichen Personengruppen durchgeführt werden.

6.7 Experimenttest und -durchführung

Um die Macken und Tücken des beschriebenen Experimentaufbau vorzeitig feststellen zu können, wurde ein Experimenttest (auch Pilottest genannt) durchgeführt. Am Experimenttest hat ein Proband teilgenommen. Bei ihm handelte es sich um einen Studenten des Diplomstudiengangs Softwaretechnik. Dieser befand sich zur Zeit des Tests unmittelbar vor dem Beginn seiner Diplomarbeit.

Der Experimenttest wurde wie geplant und ohne die Einflussnahme der Experimentatoren durchgeführt. Daraufhin wurden die Ergebnisse des Probanden analysiert und dieser von den Experimentatoren hinsichtlich der Qualität (Umfang, Schwierigkeit der Aufgaben, ...) des Experiments befragt. Die anschließende Auswertung ergab, dass das Experiment ohne größere Anpassungen durchgeführt werden konnte. Für die tatsächliche Durchführung wurden lediglich weitere Hinweise auf den Aufgabenblättern hinzugefügt.

Letztendlich wurde das getestete Experiment unter den beschriebenen Rahmenbedingungen durchgeführt. Am Experiment haben acht Studenten teilgenommen. Bei der Durchführung sind keine nennenswerten Probleme oder Planungsabweichungen aufgetreten.

6.8 Experimentergebnisse

In diesem Kapitel werden die Ergebnisse der einzelnen Phasen des Experiments vorgestellt. Die Interpretation der Ergebnisse findet in Kapitel 6.9 statt. Die Fragen des ersten und zweiten Experimentabschnitts wurden atomar ausgewertet, d. h. dass keine Punkte für teilweise richtige Antworten vergeben wurden. Eine Frage wird daher genau dann als korrekt beantwortet betrachtet, wenn deren Antwort vollständig und korrekt ist. In allen anderen Fällen ist eine Antwort falsch.

Erster Experimentabschnitt

Die Auswertung der Aufgabenblätter des ersten Experimentabschnitts ergab, dass die Probanden die gestellten Fragen (F und F')...

- ... mithilfe von UniMoDoc und der darin erstellten Moduldokumentation ($UniMoDoc$) zu 58% korrekt beantwortet haben. Insgesamt wurden 23 der 40 Fragen korrekt beantwortet.
- ... ohne UniMoDoc und der darin erstellten Moduldokumentation ($!UniMoDoc$) zu 38% korrekt beantwortet haben. Insgesamt wurden 15 der 40 Fragen korrekt beantwortet.

Im direkten Vergleich konnten somit 53% mehr Fragen mit UniMoDoc korrekt beantwortet werden.

Erste Feedbackrunde

Im Folgenden wird auf die Antworten eines Teils der Fragen eingegangen, die im ersten Feedbackbogen gestellt wurden. Der Teil umfasst alle Fragen, bei deren Antwort die Probanden aus einer vorgegebenen Menge von Antworten wählen konnten. Somit werden die Fragen, die mit Freitext beantwortet wurden, nicht behandelt. Der Grund dafür ist der, dass die Antworten dieser Fragen zunächst überprüft, gegebenenfalls gefiltert und anschließend interpretiert werden müssen. Da die Interpretation der Ergebnisse in Kapitel 6.9 beschrieben ist, findet sich auch dort die Interpretation der Freitextantworten der ersten Feedbackrunde.

Auf die Frage, ob die Probanden die Aufgaben mit oder ohne UniMoDoc besser lösen konnten, antworteten sie auf einer Skala von 1=*sehr viel besser ohne* bis 7=*sehr viel besser mit* zu...

- ... 12,5% mit 3=*besser ohne*.
- ... 25,0% mit 4=*mit und ohne gleich gut/schlecht*.
- ... 37,5% mit 6=*viel besser mit*.

... 25,0% mit 7=*sehr viel besser mit*.

Auf die Frage, ob die Probanden die Visualisierung zur Bearbeitung der Aufgaben vom Typ F-1 bis F-5 mindestens einmal eingesetzt haben, antworteten sie zu...

... 100,0% mit *Ja*.

Auf die Frage, wie schnell die Probanden mithilfe der Visualisierung die Module finden konnten, welche eine bestimmte Anforderung realisieren, antworteten sie auf einer Skala von 1=*viel langsamer* bis 5=*viel schneller* zu...

... 25,0% mit 4=*schneller*.

... 75,0% mit 5=*viel schneller*.

Auf die Frage, wie schnell die Probanden mithilfe der Visualisierung ausmachen konnten, ob eine Anforderung getestet wurde, antworteten sie auf einer Skala von 1=*viel langsamer* bis 5=*viel schneller* zu...

... 12,5% mit 3=*gleich schnell*.

... 25,0% mit 4=*schneller*.

... 62,5% mit 5=*viel schneller*.

Auf die Frage, wie schnell die Probanden mithilfe der Visualisierung ausmachen konnten, welche Testklassen eine Anforderung testen, antworteten sie auf einer Skala von 1=*viel langsamer* bis 5=*viel schneller* zu...

... 12,5% mit 3=*gleich schnell*.

... 50,0% mit 4=*schneller*.

... 37,5% mit 5=*viel schneller*.

Auf die Frage, wie schnell die Probanden mithilfe der Visualisierung ausmachen konnten, welche Testprotokolle zu den Testklassen gehören, antworteten sie auf einer Skala von 1=*viel langsamer* bis 5=*viel schneller* zu...

... 50,0% mit 3=*gleich schnell*.

... 25,0% mit 4=*schneller*.

... 25,0% mit 5=*viel schneller*.

Auf die Frage, wie schnell die Probanden mithilfe der Visualisierung ausmachen konnten, wie die Module voneinander abhängen, antworteten sie auf einer Skala von 1=*viel langsamer* bis 5=*viel schneller* zu...

... 12,5% mit 3=*gleich schnell*.

... 12,5% mit 4=*schneller*.

... 75,0% mit 5=*viel schneller*.

Auf die Frage, wie oft UniMoDoc während des Experiments abgestürzt ist, antworteten die Probanden zu...

... 100% mit *kein Mal* (0).

Auf die Frage, wie die Probanden die Benutzerfreundlichkeit von UniMoDoc werten, antworteten sie auf einer Skala von 1=*nicht benutzerfreundlich* bis 5=*sehr benutzerfreundlich* zu...

... 37,5% mit 2.

... 12,5% mit 5.

... 50,0% mit 6.

Zweiter Experimentabschnitt

Die Auswertung der digitalen Resultate des zweiten Experimentabschnitts ergab, dass die Probanden 96% der zu bearbeitenden Aufgaben innerhalb der Zeit korrekt gelöst haben. Insgesamt wurden 23 von 24 Aufgaben korrekt gelöst.

Zweite Feedbackrunde

Im Folgenden wird auf die Antworten eines Teils der Fragen eingegangen, die im zweiten Feedbackbogen gestellt wurden. Ebenso wie in den Ergebnissen der ersten Feedbackrunde wird nur auf die Fragen eingegangen, bei deren Antwort die Probanden aus einer vorgegebenen Menge von Antworten wählen konnten.

Auf die Frage, welches Werkzeug die Probanden zur Dokumentation eines Moduls verwenden würden, antworteten sie zu...

... 50,0% mit *UniMoDoc*.

... 25,0% mit *Wiki*.

... 12,5% mit *Javadoc*.

... 12,5% mit *UML Tool*.

Auf die Frage, welche Art der Moduldokumentation die Probanden vorfinden wollen würden, sofern sie eine fremde Software weiterentwickeln müssten, antworteten sie zu...

... 62,5% mit *mit UniMoDoc*.

... 32,5% mit *in einem Wiki*.

Auf die Frage, wie die Probanden ihre Java-Erfahrungen auf einer Skala von 1=*keine Erfahrung* bis 7=*sehr viel Erfahrung* einschätzen, antworteten sie zu...

... 12,5% mit 2.

... 37,5% mit 5.

... 37,5% mit 6.

... 12,5% mit 7.

Auf die Frage, wie die Probanden ihre Javadoc-Erfahrungen auf einer Skala von 1=*keine Erfahrung* bis 7=*sehr viel Erfahrung* einschätzen, antworteten sie zu...

... 25,0% mit 2.

... 37,5% mit 3.

... 25,0% mit 5.

... 12,5% mit 7.

Auf die Frage, wie oft UniMoDoc während des Experiments abgestürzt ist, antworteten die Probanden zu...

... 100% mit *kein Mal* (0).

Auf die Frage, wie die Probanden die Benutzerfreundlichkeit von UniMoDoc werten, antworteten sie auf einer Skala von 1=*nicht benutzerfreundlich* bis 7=*sehr benutzerfreundlich* zu...

... 25% der Probanden mit 3.

... 12,5% der Probanden mit 5.

... 62,5% der Probanden mit 6.

6.9 Schlussfolgerungen

In den nachstehenden Kapiteln werden die Schlüsse erläutert, welche von den Experimentatoren aus den Ergebnissen hinsichtlich der Experimentfragen gefolgert wurden. Abschließend wird das Fazit zum Experiment gezogen.

Schlussfolgerungen bezüglich EF-1

EF-1 Können die Probanden mithilfe einer in UniMoDoc erstellten Moduldokumentation Fragen zu dem Softwareprojekt, das sie dokumentiert, effizienter und effektiver beantworten, als lediglich mithilfe der Entwicklungsartefakte (ausschließlich der Moduldokumentation) des Projekts?

Im ersten Experimentabschnitt haben die Probanden mithilfe von UniMoDoc doppelt so viele Fragen korrekt beantwortet, als mithilfe der Entwicklungsartefakte. Des Weiteren konnten sie die Fragen nach eigener Aussage mit UniMoDoc besser beantworten, als ohne UniMoDoc. Einen wichtigen Beitrag dazu liefert vermutlich die Visualisierung, da es nach Angabe der Probanden mit ihr möglich war die meisten Aufgaben wesentlich schneller zu lösen. Die gestellten Fragen konnten somit mit UniMoDoc effizienter und effektiver beantwortet werden, als lediglich mithilfe der Entwicklungsartefakte. Anhand der Resultate kann die Frage EF-1 daher mit „Ja“ beantwortet werden.

Schlussfolgerungen bezüglich EF-2

EF-2 Tragen die explizit definierbaren Beziehungen und deren Visualisierung zum Verständnis der Beziehungen, die in einer Moduldokumentation existieren, bei?

Da alle Probanden die Visualisierung beim Beantworten der Fragen des ersten Experimentabschnitt verwendet haben und insgesamt mehr Fragen mithilfe von UniMoDoc korrekt beantwortet wurden, kann geschlossen werden, dass die Visualisierung und die damit verbundenen Konzepte zum Verständnis der Probanden beigetragen haben. Untermauert wird diese Schlussfolgerung von den Antworten der Probanden hinsichtlich der hilfreichen Funktionen von UniMoDoc. Im ersten Feedbackbogen nannten 87,5% der Probanden darin entweder die Visualisierung und/oder die Beziehungen. EF-2 kann auf Grundlage der Resultate mit „Ja“ beantwortet werden.

Schlussfolgerungen bezüglich EF-3

EF-3 Wird UniMoDoc während dem Experiment abstürzen?

UniMoDoc ist bei keinem der Probanden während der Experimentdurchführung abgestürzt. Daraus kann zum einen geschlossen werden, dass UniMoDoc eine grundlegende Stabilität aufweist und zum anderen, dass die Stabilität des Programms die subjektive Meinung der Probanden nicht negativ beeinflusst hat. Die Frage EF-3 ist damit klar mit „Nein“ zu beantworten.

Schlussfolgerungen bezüglich EF-4

EF-4 Empfinden die Probanden UniMoDoc bei der Bearbeitung der Aufgaben als benutzerfreundlich?

UniMoDoc wurde hinsichtlich der Benutzerfreundlichkeit mehrheitlich positiv bewertet. Auf Grundlage des zweiten Experimentabschnitts, der die Evaluation der Benutzerfreundlichkeit und Stabilität zum Ziel hatte, entschieden 50% der Probanden, dass sie UniMoDoc für die Dokumentation von Modulen verwenden würden. Entsprechend der Ergebnisse lässt sich die Experimentfrage EF-4 mit „Ja“ beantworten. Bei der Auswertung der Feedbackbögen konnte kein negativer Einfluss, welcher durch die Benutzerfreundlichkeit bedingt war, auf die Resultate der Experimentfragen EF-1 und EF-2 erkannt werden. Die Freitextkommentaren ermöglichten dennoch die Identifikation einiger Schwächen von UniMoDoc hinsichtlich der Benutzerfreundlichkeit.

Fazit

Insgesamt zeigt die Evaluation ein positives und vielversprechendes Ergebnis. In ihr wurde gezeigt, dass UniMoDoc und die darin umgesetzten Konzepte einen Mehrwert bei der Dokumentation von Modulen bieten. Gleichzeitig konnte in der Evaluation noch Verbesserungspotenzial in Bezug auf die Benutzerfreundlichkeit von UniMoDoc festgestellt werden. Um die Ergebnisse der Evaluation zu belegen und zu präzisieren, wäre ein weiterer Evaluationslauf mit einer größeren Anzahl an Probanden sinnvoll. Insbesondere wäre eine Feldstudie interessant, in welcher UniMoDoc und dessen Brauchbarkeit in einem realen Umfeld evaluiert wird.

7 Zusammenfassung und Ausblick

Dieses Kapitel fasst die wesentlichen Punkte der Diplomarbeit zusammen und gibt einen Ausblick hinsichtlich der Fortführung und Verbesserung der in dieser Arbeit entstandenen Konzepte und Programmteile von UniMoDoc.

7.1 Zusammenfassung

Das Ziel dieser Diplomarbeit bestand ursprünglich darin, ein Konzept für die Verwaltung von Testinformationen in einem Werkzeug zur Moduldokumentation zu entwickeln, das entstandene Konzept in einem funktionsfähigen Prototypen umzusetzen und diesen abschließend zu evaluieren. Ausgegangen wurde dabei von den Forschungsergebnissen von [Kircher, 2012]. Wegen der verwandten Thematik wurden die Konzepte, deren Umsetzung und Evaluation in enger Kooperation mit [Pankratz, 2013] durchgeführt. Als Resultat beider Arbeiten sollte ein Werkzeug zur Moduldokumentation entstehen.

Kapitel 2 beschrieb die Ergebnisse der Recherche, welche der Ausgangspunkt dieser Diplomarbeit war. In ihm wurde erläutert, dass Module idealerweise in der Softwarearchitekturdokumentation dokumentiert werden und für die Softwarearchitekturdokumentation bislang keine geeigneten Werkzeuge existierten. Eine Unzulänglichkeit der Werkzeuge ist beispielsweise, dass die Beziehungen, welche zwischen Modulen bestehen, nicht geeignet ausgedrückt werden können. Ausgehend von diesen Erkenntnissen haben die Autoren beschlossen, den Anwendungsbereich des zu entwickelnden Werkzeugs weiter zu fassen und anstelle eines Werkzeugs zur Moduldokumentation eines für die Softwarearchitekturdokumentation zu erstellen. Im weiteren Verlauf des Kapitels wurde der Fokus auf die Testinformationen gelegt, welche potenziell zu einem Modul festgehalten werden können. Dazu wurde untersucht, was Testinformationen sind, wo sie entstehen und wie sie dokumentiert werden. Hierbei wurde zum einen erläutert, dass zwischen den Testinformationen Beziehungen bestehen können, und zum anderen, dass für die Erstellung und Verwaltung von Testinformationen bereits etablierte und ausgereifte Werkzeuge existieren.

Kapitel 3 beschrieb zunächst die Anforderungen und die Zielgruppe des zu entwickelnden Softwarearchitekturdokumentationswerkzeugs Universal Module Documenter (UniMoDoc). Die Anforderungen wurden gemäß der Erkenntnisse aus Kapitel 2 gewählt. Neben den Anforderungen, welche die Verwaltung von Testinformationen betreffen, wurden weitere für die Verwaltung und Visualisierung der oben genannten Beziehungen formuliert. Im Kapitel wurde anschließend auf die Konzepte des Werkzeugs eingegangen. In ihm können

Dokumente und Kapitel auf Basis von Vorlagen erstellt werden. Die Bestandteile des Dokuments, zu denen auch interne Repräsentationen von externen Ressourcen zählen, lassen sich über Beziehungen miteinander verknüpfen. Jede Beziehung besitzt eine explizite Semantik, welche durch den Typ der Beziehung vorgegeben ist. Für die Beziehungen werden im Kapitel Beziehungstypen vorgeschlagen, welche für die Beziehungen der Softwarearchitekturdokumentation – insbesondere für die zu oder zwischen den Testinformationen – relevant sind. Abschließend wurden Metadaten für eine Vorlage vorgeschlagen, mit der die Testinformationen eines Moduls in UniMoDoc verwaltet werden können.

Kapitel 4 und Kapitel 5 erläuterten die Umsetzung der beschriebenen Konzepte und präsentierten die Ergebnisse der Umsetzung. Dabei wurde auch die entstandene Vorlage vorgestellt.

Kapitel 6 beschrieb die Evaluation von UniMoDoc und der darin umgesetzten Konzepte. Um die Brauchbarkeit der Konzepte zu evaluieren, wurde ein kontrolliertes Experiment mit 8 Studenten des Masterstudiengangs *Softwaretechnik* der Universität Stuttgart durchgeführt. Im Experiment wurden den Studenten in diesem Zusammenhang Fragen zu einem Softwareprojekt gestellt. Einen Teil der Fragen sollten sie mithilfe von UniMoDoc und einer darin erstellten Moduldokumentation und einen mithilfe der Entwicklungsartefakte des Projekts ausschließlich der Moduldokumentation lösen. Die Auswertung der Ergebnisse zeigte, dass UniMoDoc einen wesentlichen Vorteil bei der Beantwortung der Fragen bot. Die Ergebnisse ließen die Schlussfolgerung zu, dass der Vorteil auf die explizit definierbaren Beziehungen und deren Visualisierung zurückzuführen war.

7.2 Ausblick

In diesem Kapitel werden einige Punkte beschrieben, die in den Konzepten oder Programmteilen von UniMoDoc, welche in dieser Arbeit entstanden sind, in Zukunft verbessert oder ergänzt werden können. Diese Punkte konnten während der Umsetzung und der Vorbereitung und Auswertung der Evaluation identifiziert werden.

- Um von Widgets auf die Daten anderer Widgets zugreifen zu können, wäre es notwendig deren Sections miteinander in Beziehung zu setzen. Soll dies ermöglicht werden, muss die Klasse `Sections` erweitert werden, indem sie von der `IRelation`-Schnittstelle erbt. Da im Quellcode an einigen Stellen eine unterschiedliche Behandlung der einzelnen `IRelation`-Implementierungen notwendig ist, müssten diese Codeabschnitte gegebenenfalls angepasst werden.
- Alle Regeln des benutzerdefinierten Filters werden in der jetzigen Umsetzung über den logischen UND-Operator miteinander verknüpft (z. B. Regel1 & Regel2 & Regel3). Möchte man komplexe logische Verknüpfungen, wie z. B. Regel1 & (Regel2 | Regel3), unterstützen, müssten die Regeln mit eindeutigen IDs versehen werden. Die IDs könnten anschließend vom Benutzer dazu verwendet werden, um die Verknüpfungsvorschrift in einem Textfeld über eine vordefinierte Syntax zu definieren.

- Relation Types können hinsichtlich ihrer Relation Endpoints derzeit nicht eingeschränkt werden. Beispielsweise ist es nicht möglich, in einem Relation Type zu definieren, dass seine Relations nur zwischen Relation Endpoints vom Typ Reference erstellt werden dürfen. Um eine solche Einschränkung zu ermöglichen, müssten die Klassen `RelationType`, `RelationTypeDialog` und `RelationDialog` erweitert werden.
- Die Visualisierung könnte auf mehrere Arten erweitert bzw. verbessert werden. Zum einen könnte sie um Filterfunktionen erweitert werden, über welche bestimmte Elemente des Graphen ein- oder ausgeblendet werden können. Beispielsweise wäre eine Filterfunktion denkbar, die es ermöglicht nur solche Relations im Graph anzuzeigen, welche von einem bestimmte Relation Types sind. Zum anderen könnte die Visualisierung verbessert werden, indem dem Benutzer mehrere Layouts für die Anordnung der Knoten und Kanten angeboten werden.

Literaturverzeichnis

[Aberdour, 2013] Aberdour, M. (2013). *Open source software testing tools*. URL: http://www.opensourcetesting.org/. (Zitiert auf Seite 32)

[Beck und Gamma, 2013] Beck, K. und Gamma, E. (2013). *JUnit*. URL: http://junit.org/.

[Black, 2003] Black, A. (2003). *Critical Testing Process: Plan, Prepare, Perform, Perfect*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Zitiert auf Seite 25)

[Boehm, 1979] Boehm, B. W. (1979). *Guidelines for Verifying and Validating Software Requirements and Design Specifications*. In Samet, P. A., editor, *Euro IFIP 79*, pages 711–719. North Holland. (Zitiert auf den Seiten 7, 22 und 23)

[BSI, 1998] BSI (1998). *British Standard 7925-1, Vocabulary*. (Zitiert auf Seite 21)

[Craig, 2002] Craig, Rick D. und Jaskiel, S. P. (2002). *Systematic Software Testing*. Artech House Publishers. (Zitiert auf Seite 25)

[Frühauf et al., 2004] Frühauf, K., Ludewig, J., und Sandmayr, H. (2004). *Software-Prüfung - eine Anleitung zum Test und zur Inspektion (5. Aufl.)*. vdf. (Zitiert auf den Seiten 7 und 22)

[Garlan et al., 2010] Garlan, D., Bachmann, F., Ivers, J., Stafford, J., Bass, L., Clements, P., und Merson, P. (2010). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition. (Zitiert auf den Seiten 18, 19, 20, 36, 40, 44, 49 und 50)

[IEEE-1471, 2000] IEEE-1471 (2000). *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. *IEEE Std 1471-2000*, pages i–23. (Zitiert auf Seite 18)

[IEEE-24765, 2010] IEEE-24765 (2010). *Systems and software engineering - Vocabulary*. Technical report. (Zitiert auf den Seiten 17, 20 und 21)

[IEEE-829, 2008] IEEE-829 (2008). *IEEE Standard for Software and System Test Documentation*. *IEEE Std 829-2008*. (Zitiert auf den Seiten 20, 22, 26 und 27)

[Illes et al., 2006] Illes, T., Pohlmann, H., Roßner, T., Schlatter, A., und Winter, M. (2006). *Software-Testmanagement*. *iX Studie*. (Zitiert auf Seite 32)

[Imbus AG, 2013a] Imbus AG (2013a). *Softwaretest Werkzeuge im Überblick*. URL: http://imbus.de/tool-liste. (Zitiert auf Seite 32)

[Imbus AG, 2013b] Imbus AG (2013b). *TestBench*. URL: http://www.imbus.de/imbus-testbench/. (Zitiert auf Seite 32)

[ISTQB, 2012] ISTQB (2012). *Standard glossary of terms used in Software Testing*. (Zitiert auf Seite 21)

[JGraph, 2013a] JGraph (2013a). *jGraph*. URL: http://www.jgraph.com/. (Zitiert auf Seite 59)

[JGraph, 2013b] JGraph (2013b). *jGraph Forum*. URL: http://forum.jgraph.com//. (Zitiert auf Seite 59)

[JUNG, 2013] JUNG (2013). *JUNG - Java Universal Network/Graph Framework*. URL: http://jung.sourceforge.net/. (Zitiert auf Seite 59)

[Kircher, 2012] Kircher, M. (2012). *Integrierte Dokumentation für Software-Module*. Diplomarbeit, Universität Stuttgart. (Zitiert auf den Seiten 13 und 89)

[Kircher, 2013] Kircher, M. (2013). *J-PaD*. URL: http://www.j-pad.de/. (Zitiert auf Seite 13)

[Liggesmeyer, 2002] Liggesmeyer, P. (2002). *Software-Qualität - testen, analysieren und verifizieren von Software*. Spektrum Akadem. Verl. (Zitiert auf Seite 22)

[Ludewig und Lichter, 2007] Ludewig, J. und Lichter, H. (2007). *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag. (Zitiert auf Seite 26)

[Myers, 1979] Myers, G. J. (1979). *The Art of Software Testing*. Wiley, New York. (Zitiert auf Seite 20)

[Naveh, 2013] Naveh, B. (2013). *JGraphT*. URL: http://jgrapht.org/. (Zitiert auf Seite 59)

[OpenJDK, 2013] OpenJDK (2013). *Project Jigsaw*. URL: http://openjdk.java.net/projects/jigsaw/. (Zitiert auf Seite 18)

[OSGi, 2013] OSGi (2013). *OSGi Alliance*. URL: http://www.osgi.org. (Zitiert auf Seite 18)

[Pankratz, 2013] Pankratz, D. (2013). *Tool support for software architecture documentation*. Diplomarbeit, Universität Stuttgart. (Zitiert auf den Seiten 13, 18, 35, 37, 49, 51, 61, 69, 71, 72, 76 und 89)

[Prechelt, 2001] Prechelt, L. (2001). *Kontrollierte Experimente in der Softwaretechnik - Potenzial und Methodik*. Springer-Verlag GmbH, Heidelberg, 1. aufl. edition. (Zitiert auf den Seiten 69, 74, 80 und 81)

[Spillner et al., 2012] Spillner, A., Roßner, T., Winter, M., und Linz, T. (2012). *Praxiswissen Softwaretest, Testmanagement - Aus- und Weiterbildung zum Certified Tester: Advanced Level nach ISTQB-Standard*. dpunkt.verlag. (Zitiert auf den Seiten 7, 24, 25, 28, 31 und 32)

[Spillner, 2005] Spillner, Andreas und Linz, T. (2005). *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard*. dpunkt, Heidelberg, 3. edition. (Zitiert auf den Seiten 22 und 24)

[Starke und Hruschka, 2009] Starke, G. und Hruschka, P. (2009). *Software-Architektur kompakt angemessen und zielorientiert*. Spektrum, Akad. Verl., Heidelberg. (Zitiert auf den Seiten 12, 19, 20, 49 und 50)

Alle URLs wurden zuletzt am 27. August 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift